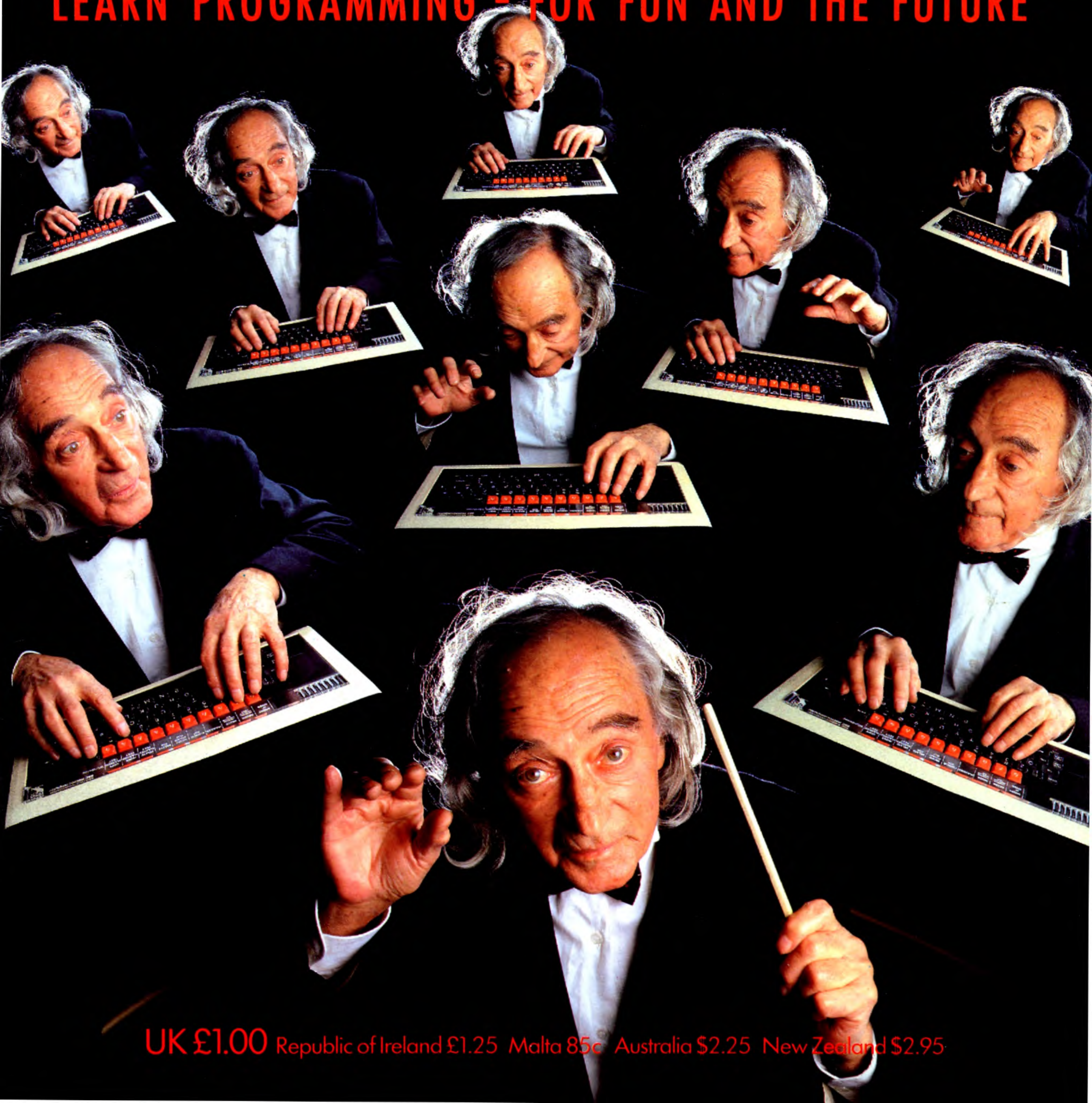
A MARSHALL CAVENDISH 44 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE



UK £1.00 Republic of Ireland £1.25 Malta 85c Australia $2.25 New Zealand $2.95

INPUT

Vol. 4 **No 44**

INDEX

The last part of INPUT, Part 52, will contain a complete, cross-referenced index. For easy access to your growing collection, a cumulative index to the contents of each issue is contained on the inside back cover.

PICTURE CREDITS
Front cover, Dave King. Pages 1365, 1366, 1368, 1370, Joe Wright. Pages 1373, 1375, Paul Chave. Page 1377, Dave King. Pages 1378, 1379, 1380, 1383, 1384, 1385, Phil Dobson. Pages 1386, 1387, 1390, 1391, Graeme Harris. Pages 1392, 1393, 1395.

Published by Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA, England. Typeset by MS Filmsetting Limited, Frome, Somerset. Printed by Cooper Clegg Web Offset Ltd, Gloucester and Howard Hunt Litho, London.

There are four binders each holding 13 issues.

HOW TO ORDER YOUR BINDERS

UK and Republic of Ireland:
Send £4.95 (inc p & p) (IR£5.95) for each binder to the address below:
Marshall Cavendish Services Ltd,
Department 980, Newtown Road,
Hove, Sussex BN3 7DN
Australia: See inserts for details, or write to INPUT, Times Consultants, PO Box 213, Alexandria, NSW 2015
New Zealand: See inserts for details, or write to INPUT, Gordon and Gotch (NZ) Ltd, PO Box 1595, Wellington
Malta: Binders are available from local newsagents.

BACK NUMBERS

Back numbers are supplied at the regular cover price (subject to availability).

UK and Republic of Ireland:
INPUT, Dept AN, Marshall Cavendish Services,
Newtown Road, Hove BN3 7DN
Australia, New Zealand and Malta:
Back numbers are available through your local newsagent.

COPIES BY POST

Our Subscription Department can supply copies to any UK address regularly at £1.00 each. For example the cost of 26 issues is £26.00; for any other quantity simply multiply the number of issues required by £1.00. Send your order, with payment to:
Subscription Department, Marshall Cavendish Services Ltd,
Newtown Road, Hove, Sussex BN3 7DN
Please state the title of the publication and the part from which you wish to start.

HOW TO PAY: Readers in UK and Republic of Ireland: All cheques or postal orders for binders, back numbers and copies by post should be made payable to:
Marshall Cavendish Partworks Ltd.

QUERIES: When writing in, please give the make and model of your computer, as well as the Part No., page and line where the program is rejected or where it does not work. We can only answer specific queries – and please do not telephone. Send your queries to INPUT Queries, Marshall Cavendish Partworks Ltd, 58 Old Compton Street, London W1V 5PA.

INPUT IS SPECIALLY DESIGNED FOR:

The SINCLAIR ZX SPECTRUM (16K, 48K, 128 and +), COMMODORE 64 and 128, ACORN ELECTRON, BBC B and B+, and the DRAGON 32 and 64.

In addition, many of the programs and explanations are also suitable for the SINCLAIR ZX81, COMMODORE VIC 20, and TANDY COLOUR COMPUTER in 32K with extended BASIC. Programs and text which are specifically for particular machines are indicated by the following symbols:

SPECTRUM 16K, 48K, 128, and + **COMMODORE 64 and 128**

ACORN ELECTRON, BBC B and B+ **DRAGON 32 and 64**

ZX81 **VIC 20** **TANDY TRS80 COLOUR COMPUTER**

DRAW IT, PRINT IT

- TYPES OF PRINTER
- INSTRUCTING THE PRINTER
- COMPATIBILITY
- SETTING UP A SCREEN
- SIDEWAYS IMAGE
- RESOLUTION AND MODE
- CONTROL CODES
- EIGHT-BIT FORMAT
- PROGRAM LOCATION
- MULTICOLOUR EFFECT

When you switch your micro off, the beautiful images you've created on the screen fade into darkness, but with a few lines of code, you can save them on paper

Your computer's graphics capabilities make it a useful tool for preparing all sorts of charts and diagrams—or making computer-generated pictures, just for fun. But whether you are interested in the practical side of computer graphics, or see it as a new artistic medium, the pictures you produce have the same great limitation—they only exist for as long as the power is on. If you want a permanent record of the image that you can file for reference, send through the post or frame as a picture, you are going to have to take it off the screen and make a hard copy.

The same problem arises, of course, when you want a paper record of a program you have written, or want to use your word-processor for sending letters. And the answer is the same—you need a printer.

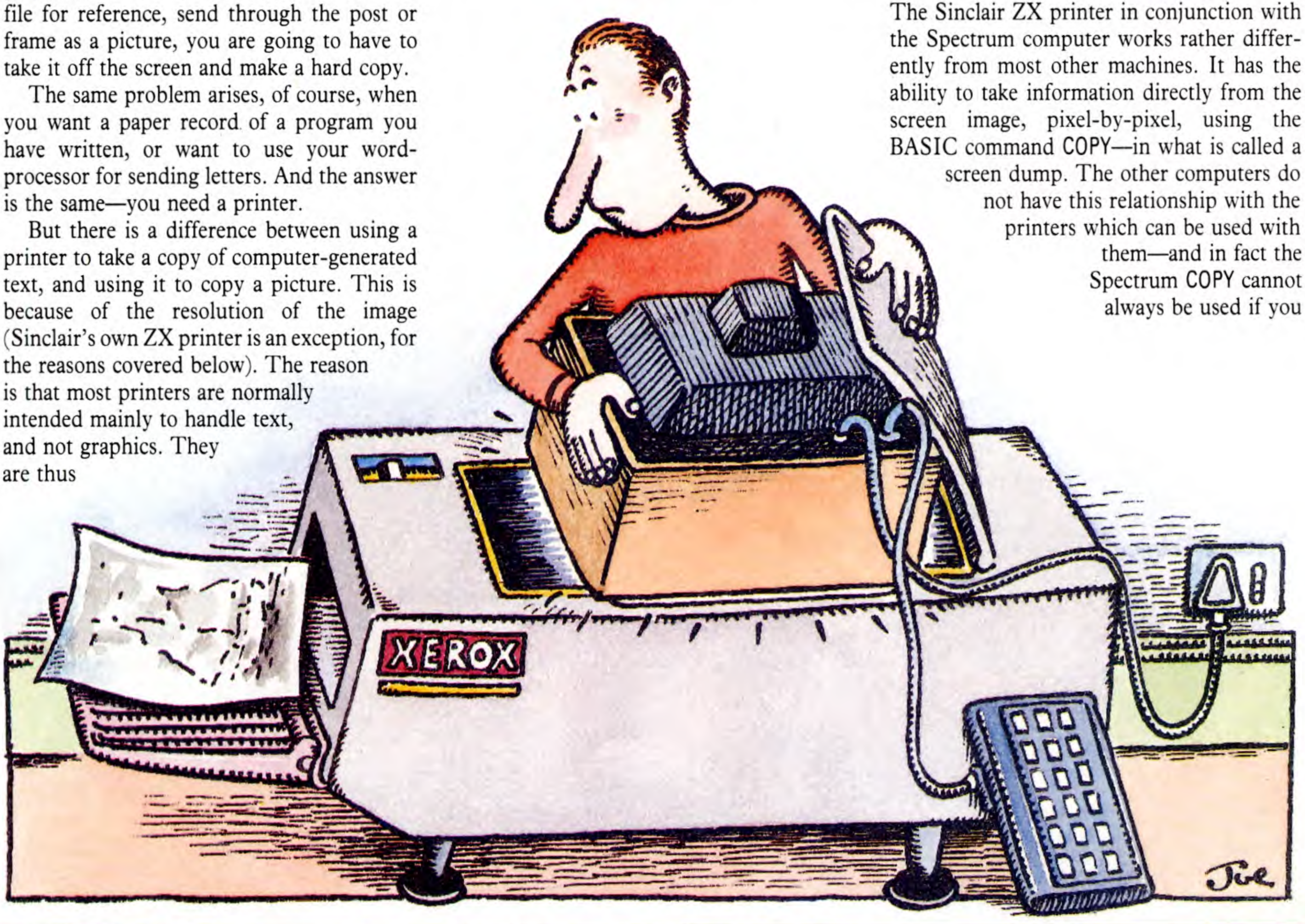
But there is a difference between using a printer to take a copy of computer-generated text, and using it to copy a picture. This is because of the resolution of the image (Sinclair's own ZX printer is an exception, for the reasons covered below). The reason is that most printers are normally intended mainly to handle text, and not graphics. They are thus set up to accept information in the form of coded numbers for each character, which they then print.

In the case of some printers, this is *all* they can do. The article on pages 225 to 229 covers the different sorts in more detail, but basically they break down into two types—those in which the type is made up of solid characters (as in a daisy-wheel printer), and those where the characters are made up of a series of dots (dot-matrix type). The daisy-wheel's ability is limited by the set of characters with which it is provided, and although it can print patterns of letters to draw a graphic image, the finest detail you can print is the size of a character square—not the sort of resolution you need to print out a typical graphics screen.

Dot-matrix printers work rather differently, however. Their characters are not preset within the machine, but are built up from a series of dots, in exactly the same way as the set of characters used by your computer is built up on the screen from individual pixels. The signal from the computer instructs the printer which pins to select in the print head, in the same way that it tells the screen which pixels to illuminate. And because the print head can reproduce an individual pixel—in detail much finer than the size of a full stop on this page—this is what makes it suitable for recording graphics screens.

INSTRUCTING THE PRINTER

The Sinclair ZX printer in conjunction with the Spectrum computer works rather differently from most other machines. It has the ability to take information directly from the screen image, pixel-by-pixel, using the BASIC command COPY—in what is called a screen dump. The other computers do not have this relationship with the printers which can be used with them—and in fact the Spectrum COPY cannot always be used if you



have hooked up another make of printer. In any of these cases you cannot key a simple, short BASIC command to ready a printer for graphics, as you can for text, because you would merely access character-sized blocks of graphics.

For example, entering ?255 on the Dragon would cause the number 255 to appear on the screen, but ?CHR$(255) would instead print a orange square—the graphics block coded 255. Now if you try to send this information to the printer, by entering ?# −2,CHR$(255) nothing would appear. The ?# −2, instructs the micro to send what follows to the printer, but this time the rest of the code specifies a DELete character, so nothing happens. Clearly, some kind of initialization routine is required to ready the printer for graphics—a special screen dump program.

This means that you must disable some of the automatic functions of the printer, and set up some others. You must ensure that the print head is set up so that the pins can be controlled individually on the pixel-by-pixel basis. And the print head and carriage advance must be set to advance a pixel at a time and not to leave spaces between the lines, rather than leaving the usual letter and line spacing with which it is programmed.

So the screen dump routine has to set up the printer in an eight-bit pin mode. This lets you send information about the screen not as ASCII codes, but as bytes—eight binary digits. You then need to disable the line feed which occurs automatically after a carriage return, using either switches on the printer or commands within the routine. Naturally, the screen dump would be useless if it didn't include some means of moving from one line to the next, so you must arrange to set the inter-line spacing and issue commands for line feed. All that remains is to generate the eight-bit codes that describe each pixel of the screen, and send these to the printer line by line.

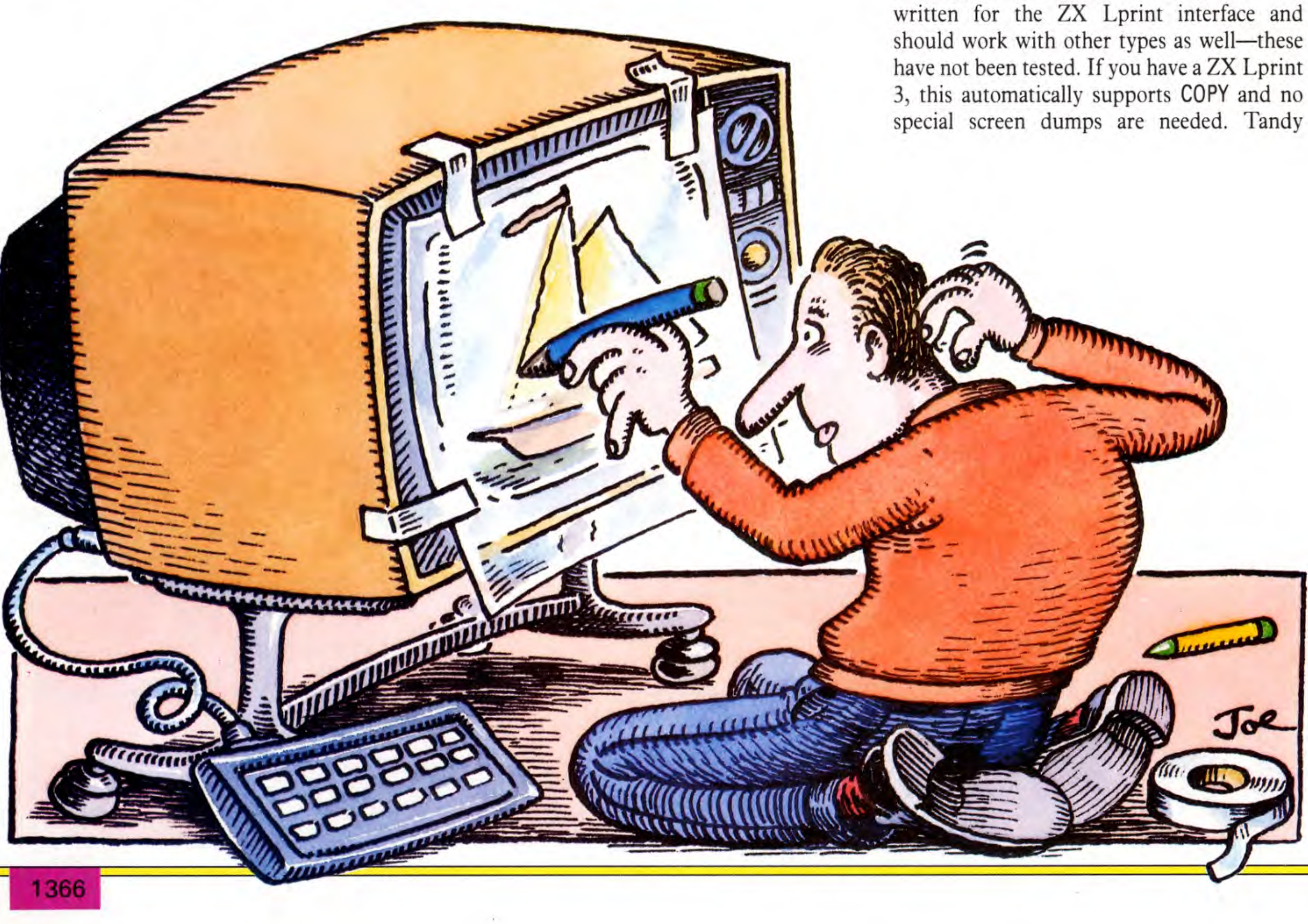
In a moment you will see a simple BASIC program to implement the process, and later you will see how to improve on this by using machine code. But first you need to think about your computer/printer combination, and how you are going to use the dump.

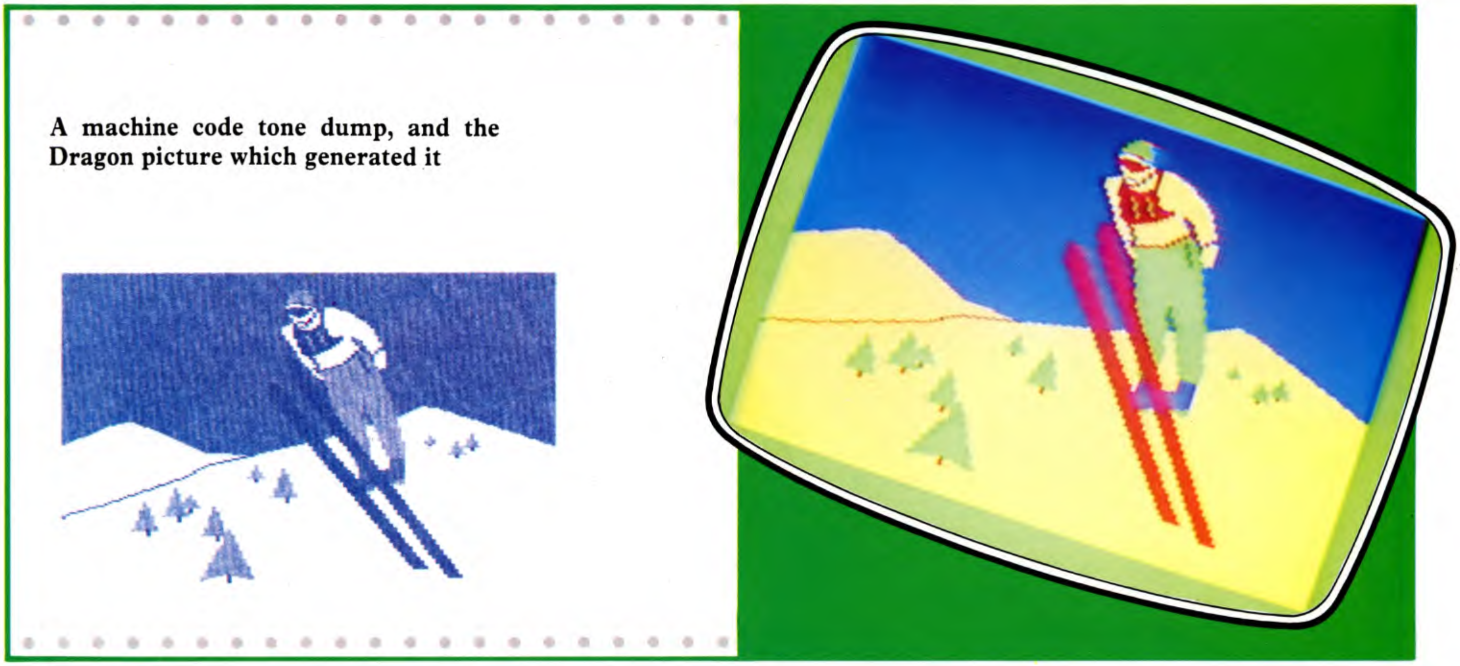
COMPATIBILITY

Any screen dump is dependent not only on the computer for which it is written, but also on the printer that is used. The programs in this article are set up for the most popular combinations.

The dumps for the Commodores are different from those for the other machines, because the printers are different. The Commodore dumps were written for a Commodore 1526 printer, (the BASIC dump) and a Commodore 1525 printer (the machine code dump). Some models in the Commodore range work in the same way, but specifications vary, so a dump to one printer will not necessarily work on another in the range. The machine code dump is unsuitable, without major changes, for the 1526 printer. The dumps for the other micros were written for an Epson FX-80 printer and should work on all Epsons and Epson-compatible machines—the most popular types used with home computers.

The Epson can also print in an eight-dot high matrix. Eight dots is the most convenient configuration, because the micro can send screen information one byte (eight bits) at a time—parallel interfacing. On the Acorns and Dragon, this is the usual method of connection for the printer (the Centronics-type interface is available as an extra for the Electron). The Spectrum needs an interface to dump to the Epson. The programs were written for the ZX Lprint interface and should work with other types as well—these have not been tested. If you have a ZX Lprint 3, this automatically supports COPY and no special screen dumps are needed. Tandy





A machine code tone dump, and the Dragon picture which generated it

owners tend to use Tandy printers which are significantly different from the Epson. The Tandy has a RS232 interface, and the dumps listed here should work via this to the Epson, but it has not been tested.

SETTING UP A SCREEN

If the program is suitable for your combination of computer, interface and printer, you are ready to try the dumps. There is no point, however, trying to dump a blank screen, so the first thing you will need is a suitable screen to dump—either one you have already SAVEd, or one created for the purpose.

The easiest way to use this is to run the program which creates the screen, and then save the screen as a file on tape or disk. Then, before you enter the screen dump, enter your own Line 1Ø as part of the dump program (this line number is deliberately not used in the dump itself) to LOAD the file back into the computer. Once this line has been obeyed and the screen has been copied, the dump will begin. Notice that on the Dragon, as explained later, the first part of the dump should come before loading the screen.

Alternatively, if you want a screen dump option built into a graphics program, you can either place the dump at the end of the screen-composing program, or treat it as a subroutine. In either case, you will need to renumber the lines, but Dragon users should keep Lines 2, 4 and 6 at the start of the program. Also, Spectrum users need to decide whether to keep the STOP in Line 11Ø (yes, if this line becomes the end of the main program, but no, if it is part of a subroutine).

SIDEWAYS IMAGE

The image dumped by this program is printed sideways, relative to the image on the screen. It is not essential to do this, but by printing the Y coordinates comfortably along the length of the paper, the printed image is, therefore, larger than the other way around.

```
15 LPRINT CHR$5;
2Ø LPRINT CHR$27;"A";CHR$8;
3Ø FOR x=Ø TO 255 STEP 4
4Ø LPRINT CHR$13;CHR$27;"*";CHR$Ø;
   CHR$96;CHR$1;
5Ø FOR y=Ø TO 175 STEP 8
6Ø FOR d=Ø TO -7 STEP -1
7Ø LET bt=(POINT(x,y-d)*128)+(POINT(x,
   y-d)*64)+(POINT(x+1,y-d)*32)+
   (POINT(x+1,y-d)*16)+(POINT(x+2,
   y-d)*8)+(POINT(x+2,y-d)*4)+
   (POINT(x+3,y-d)*2)+(POINT(x+3,
   y-d))
72 LPRINT CHR$bt;
74 LPRINT CHR$bt;
8Ø NEXT d
9Ø NEXT y
1ØØ NEXT x
11Ø LPRINT CHR$4:STOP
```

Line 15 disables the Spectrum's ASCII character set. The LPRINT in Line 2Ø instructs the computer to send what follows to the printer only and then sends an escape code A (CHR$27 followed by "A") to set up the printer to receive information about line spacing. The CHR$ 8 in this line sets the carriage advance to give no gap between the printed lines. Line 3Ø sets up a loop to step across the screen in blocks of four pixels. Line 4Ø sends carriage return (code 13), then escape code *Ø to set up the printer to receive the number of bytes that will be sent in the current line. This byte is composed of codes 96 and 1 (also in Line 4Ø), which are interpreted as 1*256 added to 96, to make 352 bytes.

Line 5Ø sets up a loop to step from bottom to top of the screen, and Line 6Ø steps through a block of eight pixels. Before all three loops are incremented, Line 7Ø reads the pixels, which are evaluated and set to the variable bt. This value is printed twice by Lines 72 and 74. The second printing is inserted so that the printed image will appear at twice the normal point size. Line 11Ø enables the Sinclair character set again.

```
2Ø OPEN5,4:OPEN4,4:OPEN6,4,6:PRINT#6,
   CHR$(2Ø)
25 X=8192
27 A(Ø)=128:A(1)=64:A(2)=32:A(3)=16:
   A(4)=8:A(5)=4:A(6)=2:A(7)=1
3Ø FORNN=1TO25
4Ø FORN=1TO4Ø:A$=""
45 FOR Z=Ø TO 7:A=Ø
5Ø FOR ZZ=Ø TO 7:IF(PEEK(X+ZZ)AND
   A(Z))=Ø THEN 7Ø
6Ø A=A+A(ZZ)
7Ø NEXT ZZ:A$=A$+CHR$(A):NEXTZ:X=
   X+8:PRINT#5,A$
8Ø PRINT#4,TAB(N)CHR$(254)CHR$(141);
9Ø NEXTN:PRINT#4:NEXTNN
1ØØ CLOSE6:CLOSE4:CLOSE5
```

The program is as for the Commodore 64, but with the following changes:

```
3Ø FOR NN=1 TO 23
4Ø FOR N=1 TO 22
```

Line 2Ø addresses the printer and sends a code (#6) to set the interline spacing so that there is no gap between the lines of print. Line 25 gives the start address of the hi-res screen memory, which is the area from which you would normally dump graphics. This address is important, because the Commodores read information about the screen from memory, and not directly by checking the attributes of pixels on the screen.

Although the screen memory normally starts at 8192, the address at which an image is stored is sometimes reset by the program which generates it. If this is the case, the new start address must be substituted for the value in Line 25. For example, the Computer Aided Design (CAD) program on page 568 lets you dump a design to a printer, but the start address is 24576. If you use a Simons' BASIC cartridge, this will also require you to specify a different starting address—57344. Furthermore, you need to switch out the Kernal ROM to be able to read the address. The box in this page gives details on finding the start address of any graphics screen.

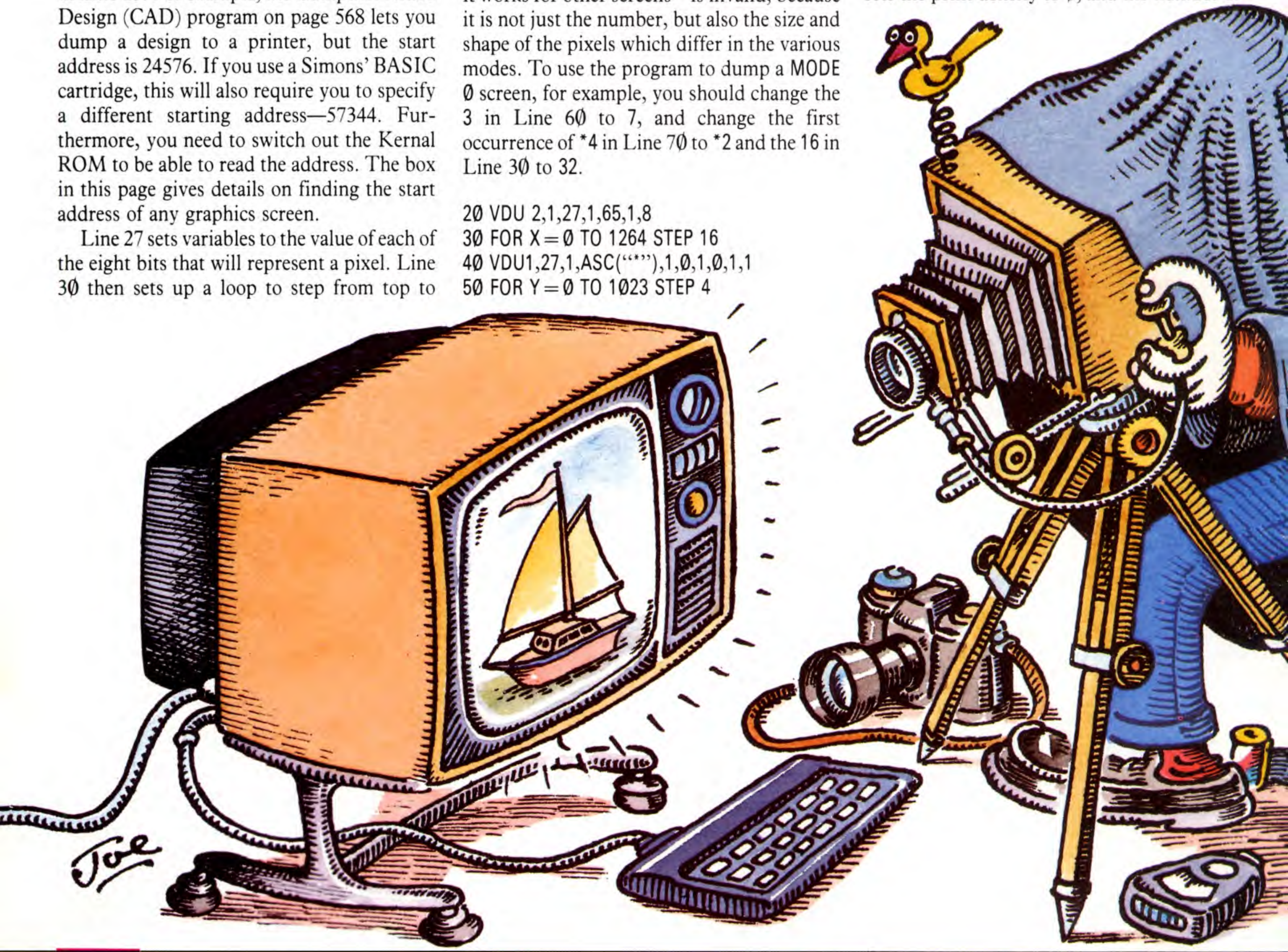
Line 27 sets variables to the value of each of the eight bits that will represent a pixel. Line 3Ø then sets up a loop to step from top to bottom of the screen, and Line 4Ø sets up a second loop to step across the screen. Line 45 loops through eight memory locations that store the screen. Line 5Ø reads the memory bit pattern, which is set to A at Line 6Ø. Line 7Ø places the bit pattern into the printer's memory, and Line 8Ø actually prints it.

An important consideration for users is the effect of mode on the resolution of the image. On Acorn computers the pixels are of different sizes depending on the graphics mode selected. The program listing which follows works only in MODE 1, which has 40 × 8 = 320 picture elements across the screen. By comparison, MODE Ø has 640 and MODE 2 has 160 elements, so allowances must be made for these differences.

Unfortunately, the obvious solution—designing a dump for MODE Ø (the screen with the largest number of pixels) and hoping it works for other screens—is invalid, because it is not just the number, but also the size and shape of the pixels which differ in the various modes. To use the program to dump a MODE Ø screen, for example, you should change the 3 in Line 6Ø to 7, and change the first occurrence of *4 in Line 7Ø to *2 and the 16 in Line 3Ø to 32.

```
2Ø VDU 2,1,27,1,65,1,8
3Ø FOR X=Ø TO 1264 STEP 16
4Ø VDU1,27,1,ASC(“*”),1,Ø,1,Ø,1,1
5Ø FOR Y=Ø TO 1Ø23 STEP 4
6Ø B%=Ø:FOR D=Ø TO 3
7Ø B%=B%*4+POINT(X+D*4,Y)
8Ø NEXT:VDU1,B%
9Ø NEXT:VDU1,13
1ØØ NEXT:VDU3
```

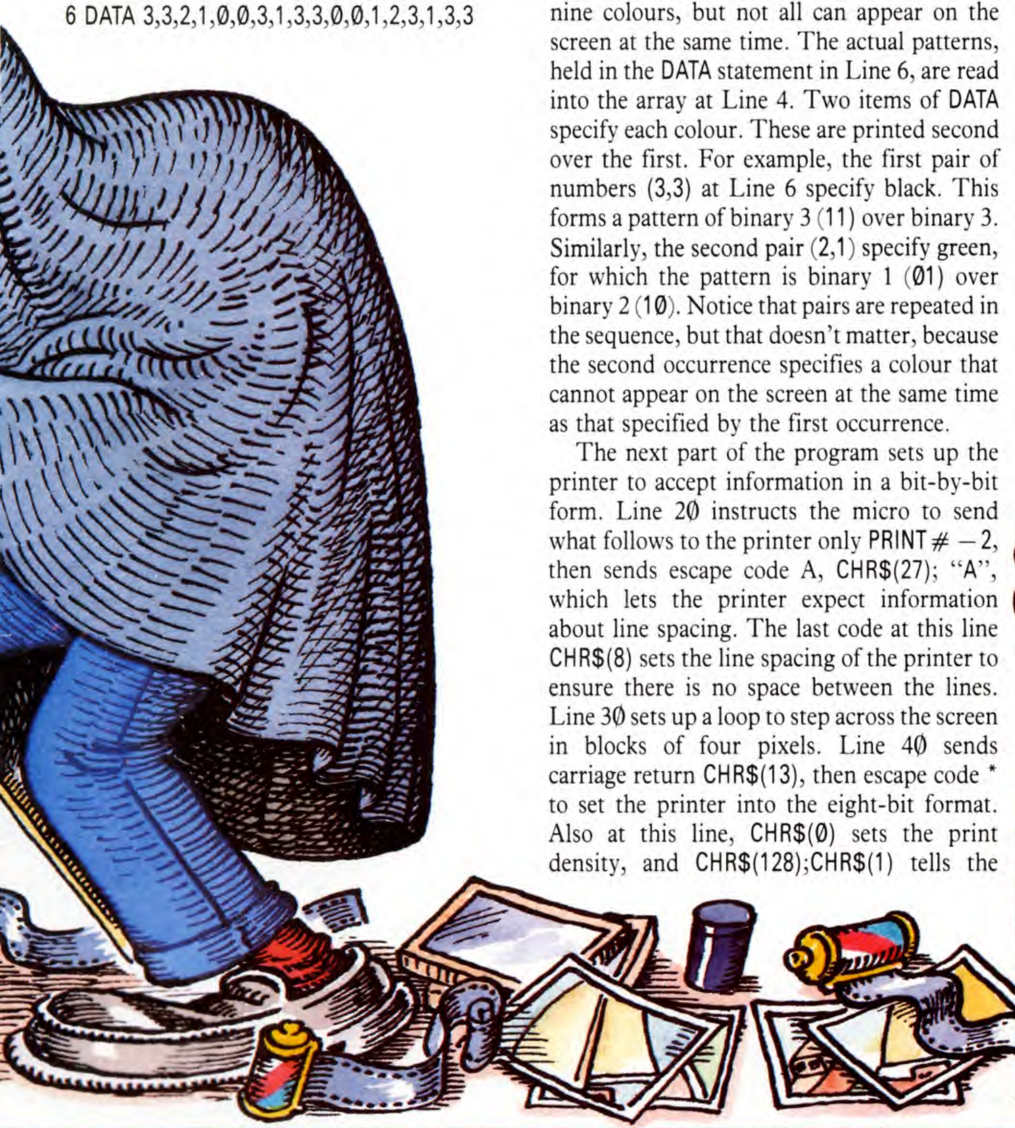
The first control code (2) after the VDU statement in Line 2Ø turns on the printer. The second code (1) sends the next number to the printer only, so 27 (the ASCII code for ESCAPE is sent. The next part—1, 65—sends A (the character whose ASCII code is 65). The result of this is to send an escape code A which tells the printer that line spacing information follows. This information is conveyed by the number 8, which ensures that the carriage advances to leave no spacing between the lines. By the same process, Line 4Ø sends ESCAPE *—a code which sets the printer into an eight-bit format—to receive data in bytes that address each pin on the print head. The next code in this line (1,Ø) sets the print density to Ø, and the number of



bytes making up each band in which the screen is scanned is given by Ø (sent as 1,Ø) and 1 (sent as 1,1), also in Line 4Ø. The Ø and 1 are interpreted as Ø added to 1*256, to make 256 bytes.

The program steps through three loops while line 7Ø reads the pixels on the screen into the variable B%, which is sent to the printer by Line 8Ø. The innermost loop (Lines 6Ø to 8Ø) steps across four pixels—the width of the scan band. The second loop (Lines 5Ø to 9Ø) steps from bottom to top of the screen through 256 pixels (1Ø24/4), and the first loop (Lines 3Ø to 1ØØ) steps across the screen through 8Ø times 4 or 32Ø pixels. Line 9Ø also sends carriage return to the printer only (1,13), and when the dump is complete, VDU 3 (Line 1ØØ) turns off the printer.

```
2 DIMA(8,1)
4 FORK=ØTO8:READA(K,Ø),A(K,1):NEXT
6 DATA 3,3,2,1,Ø,Ø,3,1,3,3,Ø,Ø,1,2,3,1,3,3
2Ø PRINT#-2,CHR$(27);“A”;CHR$(8)
3Ø FORL=ØTO255 STEP4
4Ø PRINT#-2,CHR$(13);CHR$(27);“*”;
   CHR$(Ø);CHR$(128);CHR$(1);
5Ø FORK=191TOØ STEP-1
6Ø T=Ø:S=Ø:FORM=ØTO3:P=PPOINT
   (L+M,K):T=T*4+A(P,Ø):S=S*4+A(P,
   1):NEXT
7Ø PRINT#-2,CHR$(T);CHR$(S);
8Ø NEXTK,L
9Ø PRINT#-2,CHR$(27);“@”
```



It is important that the first three lines of this program precede the entry of the screen which you want to dump. You should then set up the graphics mode and screen type at Line 1Ø. Because of the way that the Dragon screen is stored, the dump needs to take account of the colours used. For a detailed explanation of the complexities of the Dragon graphics screen, see page 248.

Line 2 DIMensions an array of variables to store the dot pattern of each colour. There are nine colours, but not all can appear on the screen at the same time. The actual patterns, held in the DATA statement in Line 6, are read into the array at Line 4. Two items of DATA specify each colour. These are printed second over the first. For example, the first pair of numbers (3,3) at Line 6 specify black. This forms a pattern of binary 3 (11) over binary 3. Similarly, the second pair (2,1) specify green, for which the pattern is binary 1 (Ø1) over binary 2 (1Ø). Notice that pairs are repeated in the sequence, but that doesn't matter, because the second occurrence specifies a colour that cannot appear on the screen at the same time as that specified by the first occurrence.

The next part of the program sets up the printer to accept information in a bit-by-bit form. Line 2Ø instructs the micro to send what follows to the printer only PRINT#-2, then sends escape code A, CHR$(27); “A”, which lets the printer expect information about line spacing. The last code at this line CHR$(8) sets the line spacing of the printer to ensure there is no space between the lines. Line 3Ø sets up a loop to step across the screen in blocks of four pixels. Line 4Ø sends carriage return CHR$(13), then escape code * to set the printer into the eight-bit format. Also at this line, CHR$(Ø) sets the print density, and CHR$(128);CHR$(1) tells the printer to expect 128+(1*256), or 384 bytes. From the range at Line 5Ø, you can see that the screen is 192 pixels deep, and since each pixel is sent as a 2 × 2 dot image, 2*192 gives 384.

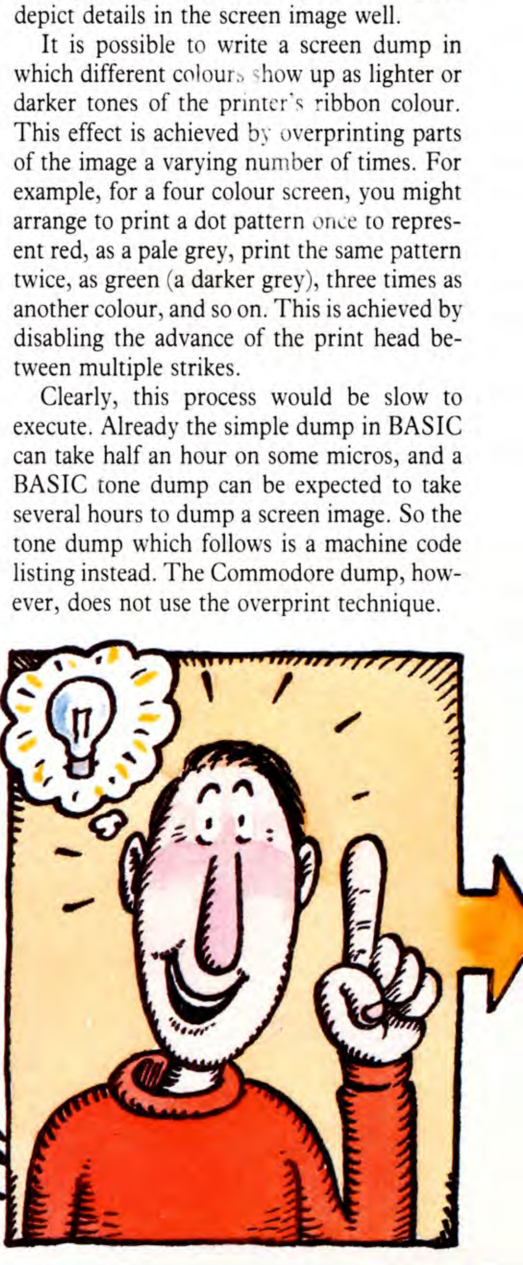
Line 5Ø loops from bottom to top of the screen, then Line 6Ø moves four pixels along, reading the screen colours and calculating the values in S and T. Line 7Ø outputs these values and Line 9Ø reinitializes the printer.

MULTICOLOUR EFFECT

The BASIC screen dump listed above gives an on/off representation of the screen image, because it reproduces the pixel pattern without differentiating the colours in the screen image. This method gives an image of even density (except on the Acorn) which does not depict details in the screen image well.

It is possible to write a screen dump in which different colours show up as lighter or darker tones of the printer's ribbon colour. This effect is achieved by overprinting parts of the image a varying number of times. For example, for a four colour screen, you might arrange to print a dot pattern once to represent red, as a pale grey, print the same pattern twice, as green (a darker grey), three times as another colour, and so on. This is achieved by disabling the advance of the print head between multiple strikes.

Clearly, this process would be slow to execute. Already the simple dump in BASIC can take half an hour on some micros, and a BASIC tone dump can be expected to take several hours to dump a screen image. So the tone dump which follows is a machine code listing instead. The Commodore dump, however, does not use the overprint technique.



Microtip

Commodore screen address

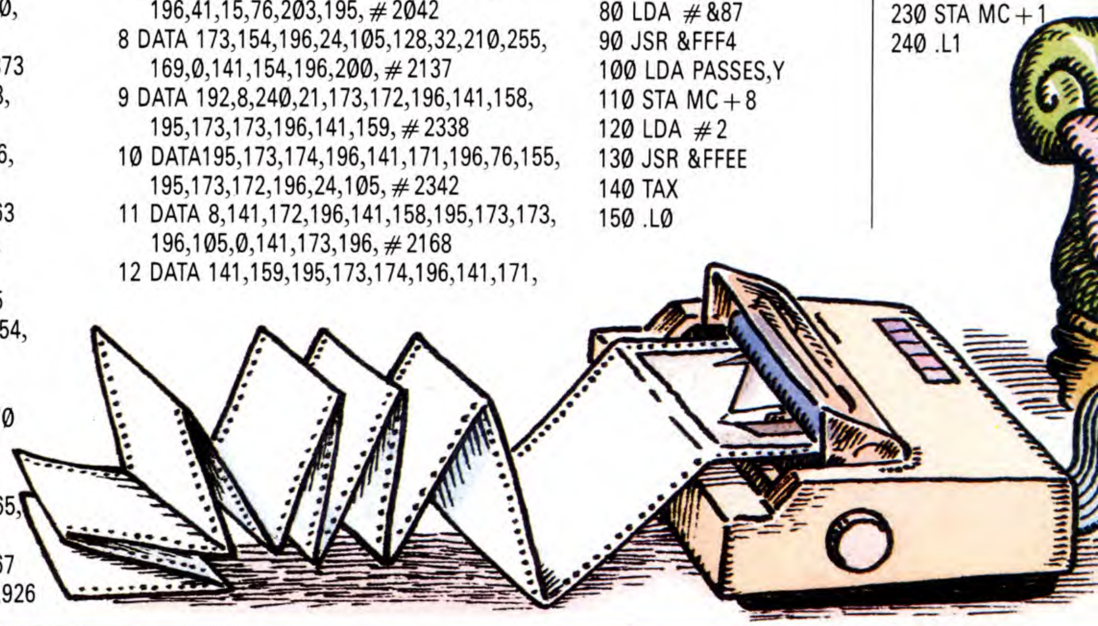
To be able to dump a screen, you must specify the address in memory where the screen starts. Usually, this address is 8192, but in some programs, the screen memory is relocated, so the new address must be specified. To find any screen's start address, enter (PEEK (53272) AND 24Ø) * 64. Sometimes the start address appears at the start of the program, often as a number multiplied by 256, to give the address.

```
1Ø CLEAR 59999
2Ø LET L=1ØØ:RESTORE L:FOR N=6ØØØØ
   TO 6Ø247 STEP 8
3Ø LET T=Ø:FOR M=Ø TO 7
4Ø READ A:LET T=T+A:POKE
   N+M,A:NEXT M
5Ø READ A:IF A< >T THEN PRINT "DATA
   ERROR IN LINE□";L:STOP
6Ø LET L=L+1Ø:NEXT N:STOP
1ØØ DATA 243,62,3,2Ø5,1,22,33,7Ø,639
11Ø DATA 235,6,4,2Ø5,9,235,62,Ø,756
12Ø DATA 5Ø,83,235,62,Ø,5Ø,84,235,799
13Ø DATA 6,175,221,33,85,235,197,62,
   1Ø14
14Ø DATA 4,237,75,83,235,245,2Ø5,15,
   1Ø99
15Ø DATA 235,245,2Ø5,3Ø,235,241,126,32,
   1349
16Ø DATA 6,2Ø3,63,2Ø3,63,2Ø3,63,23Ø,
   1Ø34
17Ø DATA 7,214,7,237,68,221,119,Ø,873
18Ø DATA 221,35,241,12,61,32,222,58,
   882
19Ø DATA 84,235,6Ø,5Ø,84,235,193,16,
   957
2ØØ DATA 2Ø5,6,7,197,33,74,235,6,763
21Ø DATA 9,2Ø5,9,235,221,33,85,235,
   1Ø32
22Ø DATA 6,175,197,6,4,3Ø,Ø,197,615
23Ø DATA 2Ø3,35,2Ø3,35,221,126,Ø,254,
   1Ø77
24Ø DATA Ø,4Ø,7,221,53,Ø,62,3,386
25Ø DATA 24,2,62,Ø,131,95,221,35,57Ø
26Ø DATA 193,16,228,123,245,215,
   241,215,1476
27Ø DATA 193,16,215,193,16,197,33,65,
   928
28Ø DATA 235,6,5,2Ø5,9,235,62,1Ø,767
29Ø DATA 215,58,83,235,198,4,5Ø,83,926
3ØØ DATA 235,21Ø,115,234,62,4,215,251,
   1326
31Ø DATA 2Ø1,126,35,215,16,251,2Ø1,197,
   1242
32Ø DATA 2Ø5,17Ø,34,71,4,126,2Ø3,7,
   82Ø
33Ø DATA 16,252,23Ø,1,193,2Ø1,197,62,
   1152
34Ø DATA 175,144,23Ø,248,71,88,22,Ø,
   978
35Ø DATA 2Ø3,35,2Ø3,18,2Ø3,35,2Ø3,18,
   918
36Ø DATA 121,2Ø3,63,2Ø3,63,2Ø3,63,111,
   1Ø3Ø
37Ø DATA 38,Ø,25,17,Ø,88,25,193,386
38Ø DATA 2Ø1,27,64,27,65,8,5,27,424
39Ø DATA 65,8,27,65,Ø,13,27,42,247
4ØØ DATA Ø,96,1,Ø,Ø,48,48,193,386
```

The start address is 8192. The two 32s printed in bold should be changed if the start address is different. To find the new number, divide the start address by 256.

```
Ø DATA 169,4,162,4,16Ø,255,32,186,255,32,
   192,255,162,4,32,#19Ø4
1 DATA 2Ø1,255,169,8,32,21Ø,255,169,Ø,141,
   171,196,141,175,196,#2319
2 DATA 141,158,195,141,172,196,169,32,141,
   173,196,141,159,195,169,#2378
3 DATA Ø,141,17Ø,196,173,171,196,141,174,
   196,173,158,195,141,176,#24Ø1
4 DATA 196,141,172,196,173,159,195,141,177,
   196,141,173,196,16Ø,Ø,#2416
5 DATA 162,Ø,173,Ø,32,57,155,196,2Ø1,Ø,
   24Ø,1Ø,173,154,196,#1749
6 DATA 24,125,163,196,141,154,196,32,1Ø6,
   196,232,224,7,2Ø8,228,#2232
7 DATA 173,175,196,2Ø1,28,2Ø8,8,173,154,
   196,41,15,76,2Ø3,195,#2Ø42
8 DATA 173,154,196,24,1Ø5,128,32,21Ø,255,
   169,Ø,141,154,196,2ØØ,#2137
9 DATA 192,8,24Ø,21,173,172,196,141,158,
   195,173,173,196,141,159,#2338
1Ø DATA195,173,174,196,141,171,196,76,155,
   195,173,172,196,24,1Ø5,#2342
11 DATA 8,141,172,196,141,158,195,173,173,
   196,1Ø5,Ø,141,173,196,#2168
12 DATA 141,159,195,173,174,196,141,171,
   196,238,17Ø,196,173,17Ø,196,#2689
13 DATA 2Ø1,4Ø,2Ø8,13Ø,169,13,32,21Ø,255,
   173,174,196,141,171,196,#23Ø9
14 DATA 173,176,196,141,158,195,173,177,
   196,141,159,195,162,Ø,32,#2274
15 DATA1Ø6,196,232,224,7,2Ø8,248,173,158,
   195,141,172,196,173,159,#2588
16 DATA 195,141,173,196,173,171,196,141,
   174,196,238,175,196,173,175,#2713
17 DATA 196,2Ø1,29,24Ø,3,76,124,195,169,15,
   32,21Ø,255,169,13,#1927
18 DATA 32,21Ø,255,32,174,255,169,4,32,195,
   255,96,238,171,196,#2314
19 DATA 173,171,196,41,7,2Ø1,Ø,24Ø,18,173,
   158,195,24,1Ø5,1,#17Ø3
2Ø DATA 141,158,195,173,159,195,1Ø5,Ø,141,
   159,195,96,173,158,195,#2243
21 DATA 24,1Ø5,57,141,158,195,173,159,195,
   1Ø5,1,141,159,195,96,#19Ø4
22 DATA Ø,128,64,32,16,8,4,2,1,1,2,4,8,16,32,
   #318
23 DATA 64,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,255,253,255,255,
   255,255,#1592
24 S=5ØØØØ:FOR Z=ØTO23:T=Ø:FOR
   ZZ=1TO15:READX:T=T+X:POKE S,X:
   S=S+1:NEXT ZZ
25 READA$:IF LEFT$(A$,1)< >"#" OR
   VAL(RIGHT$(A$,LEN(A$)-1))< >T
   THEN 27
26 NEXT Z:PRINT"DATA OK":END
27 PRINT"ERROR IN LINE"Z:END
```

```
1Ø MC=&9ØØ
2Ø PASSES=MC+9
3Ø MODE1
4Ø FOR T=Ø TO 3 STEP 3
5Ø P%=MC+16
6Ø [OPT T
7Ø .DUMP
8Ø LDA #&87
9Ø JSR &FFF4
1ØØ LDA PASSES,Y
11Ø STA MC+8
12Ø LDA #2
13Ø JSR &FFEE
14Ø TAX
15Ø .LØ
16Ø LDA DTA,X
17Ø JSR PRT
18Ø DEX
19Ø BPL LØ
2ØØ LDA #&FC
21Ø STA MC
22Ø LDA #4
23Ø STA MC+1
24Ø .L1
```

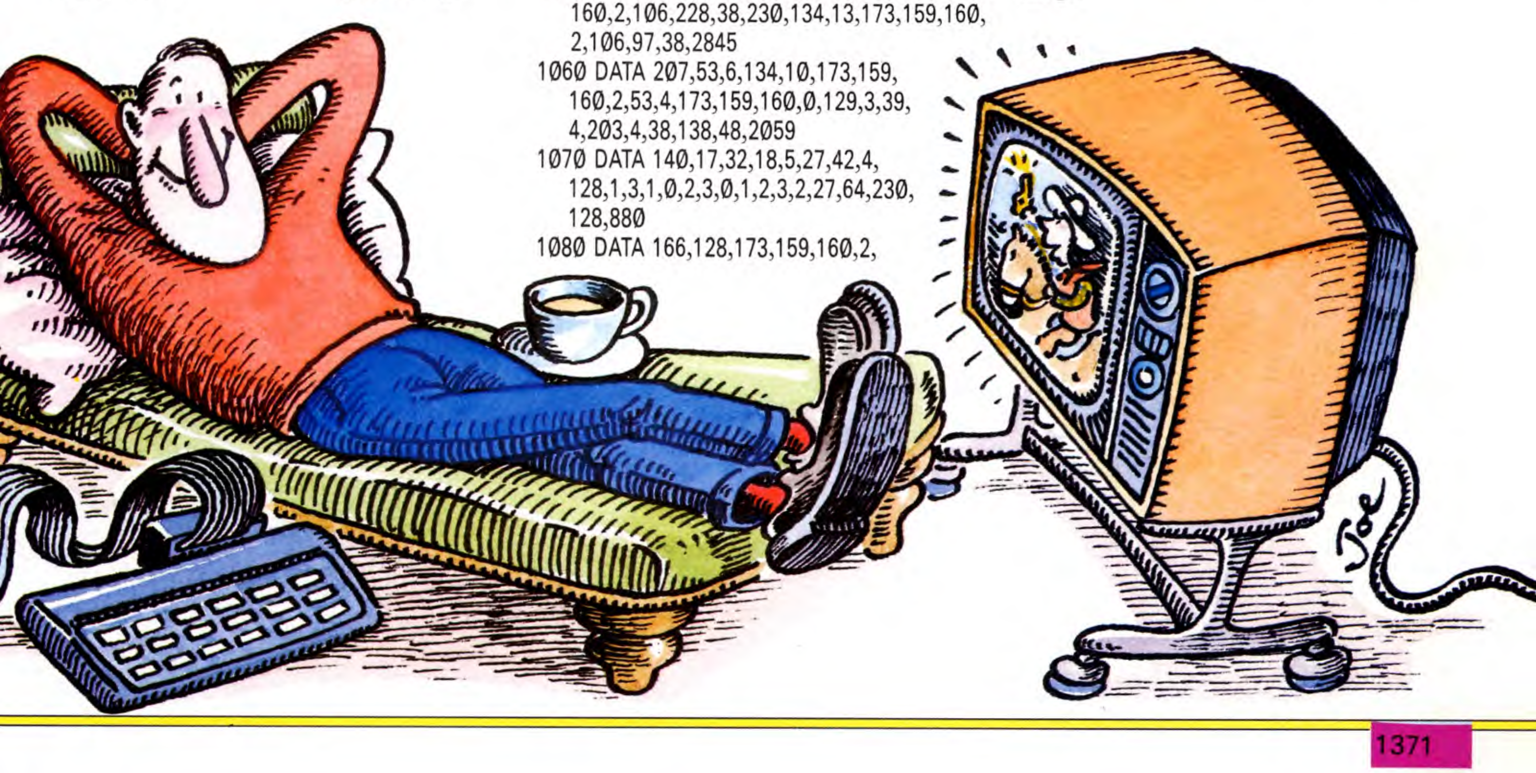


```
250 LDA MC+8
260 STA MC+7
270 LDA MC+1
280 BPL L2
290 LDA #3
300 JMP &FFEE
310 .L2
320 LDX #7
330 .L3
340 LDA DTA+3,X
350 JSR PRT
360 DEX
370 BPL L3
380 LDA #&FC
390 STA MC+2
400 LDA #3
410 STA MC+3
420 .L35
430 LDA #8
440 STA MC+6
450 .L4
460 LDA #9
470 LDX #MC□MOD 256
480 LDY #MC□DIV 256
490 JSR &FFF1
500 LDA MC+4
510 LDY MC+8
520 CPY #7
530 BNE L45
540 LSR A
550 .L45
560 CMP MC+7
570 ROL MC+5
580 SEC
590 LDA MC
600 SBC #4
610 STA MC
620 BCS L5
630 DEC MC+1
640 .L5
650 DEC MC+6
660 BNE L4
670 LDA MC+5
680 JSR PRT
690 SEC
700 LDA MC+2
710 SBC #4
720 STA MC+2
730 BCS L7
740 DEC MC+3
750 BPL L7
760 DEC MC+7
770 BNE L6
780 LDA #10
790 JSR PRT
800 JMP L1
810 .L6
820 JSR L8
830 JMP L2
840 .L7
850 JSR L8
860 JMP L35
870 .L8
880 CLC
890 LDA MC
900 ADC #32
910 STA MC
920 BCC L9
930 INC MC+1
940 .L9
950 RTS
960 .PRT
970 PHA
980 LDA #1
990 JSR &FFEE
1000 PLA
1010 JMP &FFEE
1020 RTS
1030 .DTA
1040 ]NEXT
1050 FOR T=Ø TO 1Ø:READ P%?T:NEXT
1060 DATA 8,65,27,1,Ø,Ø,42,27,9,9,13
1070 FOR T=PASSES□TO PASSES+6:
    READ ?T:NEXT
1080 DATA 1,3,7,1,1,3,1
1090 *SAVE GDUMP 900 9DE 910
```

```
10 CLEAR200,29992
20 CLS:FORK=ØTO12:T=Ø:FORL=ØTO23:
    READA
30 POKE29993+24*K+L,A:T=T+A
40 NEXT:READA:IF A< >T THENPRINT
    "DATA ERROR IN LINE";1000+10*K:END
50 NEXT
1000 DATA Ø,Ø,Ø,3,27,51,24,134,254,151,
    111,111,14Ø,242,111,14Ø,24Ø,15Ø,182,
    133,1,39,3,1Ø8,2355
1010 DATA 14Ø,231,77,39,4,129,2,38,3,1Ø8,
    14Ø,22Ø,15Ø,193,68,167,14Ø,216,48,14Ø,
    214,23,Ø,142,2632
1020 DATA 95,52,4,51,141,1,1,198,191,166,
    228,52,6,134,4,52,2,23,Ø,134,48,14Ø,1Ø7,
    166,1996
1030 DATA 134,167,192,1Ø8,97,1Ø6,228,38,
    24Ø,53,2,53,6,166,228,9Ø,193,255,38,223,
    198,3,52,6,2876
1040 DATA 51,141,Ø,212,198,192,231,228,48,
    14Ø,65,141,81,79,198,4,72,72,1Ø6,192,43,
    2,138,3,2637
1050 DATA 9Ø,38,245,173,159,16Ø,2,173,159,
    16Ø,2,1Ø6,228,38,23Ø,134,13,173,159,16Ø,
    2,1Ø6,97,38,2845
1060 DATA 2Ø7,53,6,134,1Ø,173,159,
    16Ø,2,53,4,173,159,16Ø,Ø,129,3,39,
    4,2Ø3,4,38,138,48,2Ø59
1070 DATA 14Ø,17,32,18,5,27,42,4,
    128,1,3,1,Ø,2,3,Ø,1,2,3,2,27,64,23Ø,
    128,88Ø
1080 DATA 166,128,173,159,16Ø,2,
    9Ø,38,247,57,134,32,1Ø9,141,255,48,39,1,
    68,52,2,166,1Ø1,214,2582
1090 DATA 182,193,1,34,1,68,23Ø,224,61,
    211,186,31,1,23Ø,99,84,84,84,1Ø9,141,
    255,18,39,1,2567
1100 DATA 84,58,166,99,1Ø9,141,255,8,39,8,
    132,15,64,139,15,68,32,5,132,7,64,139,7,
    198,1984
1110 DATA 1,74,43,3,88,32,25Ø,1Ø9,141,254,
    238,39,14,52,4,197,85,39,5,88,235,224,32,
    3,225Ø
1120 DATA 84,235,224,52,4,166,132,164,224,
    84,37,3,68,32,25Ø,171,141,254,2Ø6,171,
    141,254,2Ø3,57,3357
```

Although the machine code (except for the Acorn version) is in the form of a list of DATA numbers, there are checksums to guard against copying errors. But you should still observe the usual precaution before RUNning this machine code program—save a copy to tape or disk, in case the program crashes. Now RUN the program, then prepare the screen you wish to dump. To commence dumping, enter RANDOMIZE USR 60000 (for the Spectrum), SYS 50000 (for the Commodore), CALL&910 (for the Acorns) or EXEC 30000 (for the Dragon and Tandy). Notice that the Acorn program works only in the graphics modes—MODEs Ø, 1, 2, 4 or 5.

It is important that the automatic line feed is turned off on your printer before you start. The printer user manual has instructions on how to do this. For best results, fit a *well used* ribbon to your printer. This gives a greater contrast between light and dark areas of the image.



WARGAMING: MILITARY INTELLIGENCE

Turn your computer opponent from Ethelred the Unready into Genghis Khan with these additions to Cavendish Field. Stronger strategies result from using heuristics

Your completed wargame probably doesn't offer too much of a challenge at the moment. The program only allows random movement by the computer, so it's quite simple to outwit the machine after you've played a few games. Virtually any simple but sensible strategy will be effective against the computer simply because the computer has its plan, and doesn't respond to the player's tactics.

When you become bored with winning every time, the thing to do is to make the computer's game stronger. This means that you have to program in new routines. But as one of the problems with writing a program like Cavendish Field is coping with memory restrictions—particularly on the Acorn machines—any additions to the program have to be simple but effective.

A STRONGER OPPONENT

Two approaches can be used to make the computer harder to beat. The easy option is to accept the weakness of the computer's tactics and give it stronger forces than the player. This is an approach that has been adopted in some commercial wargames because this is by far the simplest to program.

If you want to see the effect of this simple approach, it can be added to the program with just one or two lines, as follows. In a moment, you'll see how to tackle a more sophisticated answer to the problem. This involves different additions, so you may want to read on first, or delete the 'extra strength' lines after you have tried them.

In the Spectrum, Dragon and Tandy programs, the unit's initial strength is held in element 6 of the troop array, and in the others it's held in element 5. To increase the computer's unit strength you only need a simple addition:

```
665 IF j=8 THEN LET T(j+i,6)=T(j+i,6)+
   FN r(1ØØ): LET T(j+i,7)=T(j+i,6)
```

```
655 IF J=8 THEN T(I+J,5)=
   T(I+J,5)+FNR(1ØØ)
```

```
65Ø IF j=8 THEN T%(i+j,5)=T%(i+j,5)+
   RND(1ØØ):T%(i+j,6)=T%(i+j,5)
```

```
665 IF J=8 THEN T(J+I,6)=T(J+I,6)+
   RND(1ØØ):T(J+I,7)=T(J+I,6)
```

The program adds a random factor to the initial strength and stores it in the current strength element of the troop array.

MILITARY INTELLIGENCE

Taking on an opponent embodying brute force and ignorance is not as satisfying as taking on an opponent, with evenly matched forces, that is intelligent. But adding intelligence to the computer's play is a far more difficult problem than adding strength.

The ideas behind adding intelligence to Cavendish Field are somewhat different from those you have seen in Othello (starting on pages 980 to 984) and Fox and Geese (starting on pages 1096 to 1100).

Consider the problem the computer has to solve: in the two board games the play has been very strictly defined. In both there is a clear way forward, having strict ways of judging success, either through tree searching or simpler ways of looking at the board. In both games the program had a strict *algorithm* built in with no uncertainty or random element. In the case of Fox and Geese, at the higher levels, the algorithm is extremely effective and should give most people a strong challenge. In the case of Othello, the algorithm is much simpler, giving the player much less of a challenge.

Returning to the wargame, you'll soon find that an algorithm is almost impossible to construct. There isn't a clearly defined way forward for either side—in fact, it is here that the differences between chess and wargaming become apparent. In chess, there are no random elements, and everything that has to be considered is connected with moves and

- A BETTER OPPONENT
- INTELLIGENCE OR STRENGTH?
- USING HEURISTICS
- SENSIBLE STRATEGIES
- THE NEW ROUTINES

spatial relationships on the board. In the case of a wargame, there is almost certainly going to be a random element built in, and there are several different sorts of variables to be considered—armour, morale, movement capabilities, numbers and so on, all play a part in the success or failure of an army. Some variables are interdependent, some are not. And, more importantly, no-one is sure what wins a battle. You cannot say 'If I follow such-and-such a course I will win.'

From a knowledge of how your wargame has been written, and the assumptions made, it could be possible to construct an algorithm, but it would be very, very complex, making the resulting programming large and slow. In practice, the answer to the problem of having a large, unfavourable, or non-existent algorithm is to resort to *heuristics*.

A heuristic is simply a 'rule of thumb' which seems likely to work in some situations. It is important to realise that heuristics are not guaranteed to work, and may even make a disastrous error on some occasions. The rule, though, is generally worth trying. In fact, heuristics are precisely what everyone uses in everyday life because of the complexity of many of the problems we are faced with.

The program, then, should have a list of heuristics built in. They could look like this:

- If you are a unit of archers, and there is an enemy unit within range, then fire.
- If you are involved in a battle, but are losing, then try going the opposite way.
- If you are near an enemy unit which is stronger than you, move away from it.
- If you are near a unit which is weaker than you, attack it.

In the last case, you will see that the computer has an advantage over you since it knows your exact strength, while you do not know its strength. However, it is worth allowing the computer to use this as it allows the machine more power of decision.

AN INTELLIGENT APPROACH

Enough of the theory—the computer needs an overall plan. No battle is fought without some kind of plan of action. The plan will depend on the victory conditions for the game—if you could win by killing the opposing general, then one possible plan would be to send all units against the general's headquarters. In Cavendish Field, victory can only be achieved by killing more of the enemy than he kills of you, so the computer wants a plan which will achieve this.

One simple method of trying to destroy enemy troops is to attack weak units with strong forces. This means that a plan saying 'Only attack an opponent who is weaker than you' could be employed. In fact, this is actually quite a complex tactic to program. It can be simplified by saying 'concentrate all your forces on one square'. By doing this (and assuming that the player doesn't concentrate on the same square), the computer's forces should outnumber the player's. A repercussion of all this is that the player's forces will outnumber the computer's elsewhere—this is where the heuristic falls down—so the other, outnumbered forces should avoid combat.

So that the player cannot readily anticipate the computer it must have a number of choices open to it. Again, to keep it simple, you can make the chosen square one occupied by one of the player's units. There is a routine each turn which tests to see if the chosen square is to be changed. This should keep the player guessing as to the computer's exact plan, and keep the computer's forces following the same pattern.

The second aspect of play which must be controlled intelligently is the individual action of each unit. Irrespective of the overall plan, each unit must respond to the immediate situation on the battlefield around it. A unit shouldn't, for example, make straight for the concentration point if many enemy units block its way.

When the computer issues its orders, it makes a series of tests. They are carried out in a carefully considered order, bearing in mind two considerations. Firstly, the tests should be conducted as rapidly as possible. This means that it should be possible to end the test routine the moment a decision has been made, without the need to continue with unnecessary tests. Secondly, the most important tests should come first. If this is not the case, the computer might make a decision according to a less important criterion.

The rules used in the program are as follows, with the most important first:

- If the unit is in combat and won last turn, then continue with the same order. No other rules will be tested if this is true.
- If the unit is in combat, but lost last turn, then it should move away from its opponent. This will not always be possible because the square the unit should move into may be already occupied, or it may be at the map edge.
- If the unit consists of archers and a target is within range, it should fire. Putting the rule here ensures that the archers will always take the less risky option of firing rather than taking part in hand-to-hand combat.
- If there is an enemy unit which is within maximum movement distance, and the unit is weaker than you, attack that unit. If, however, the unit is stronger than you, then retreat. This is a long search because enemy units on the field must be considered.
- Move towards the concentration square.

Even these few rules take a fair time to operate and, if they are all tested for each unit, the game can slow down considerably. Unfortunately, this is the penalty for programming intelligence in BASIC, as you saw in Fox and Geese—but don't worry, there are none of the huge waits you may have experienced in that game.

ADDITIONAL ROUTINES

Before adding the new routines you should delete Lines 177Ø to 179Ø as these now become a separate routine.

Now add these lines:

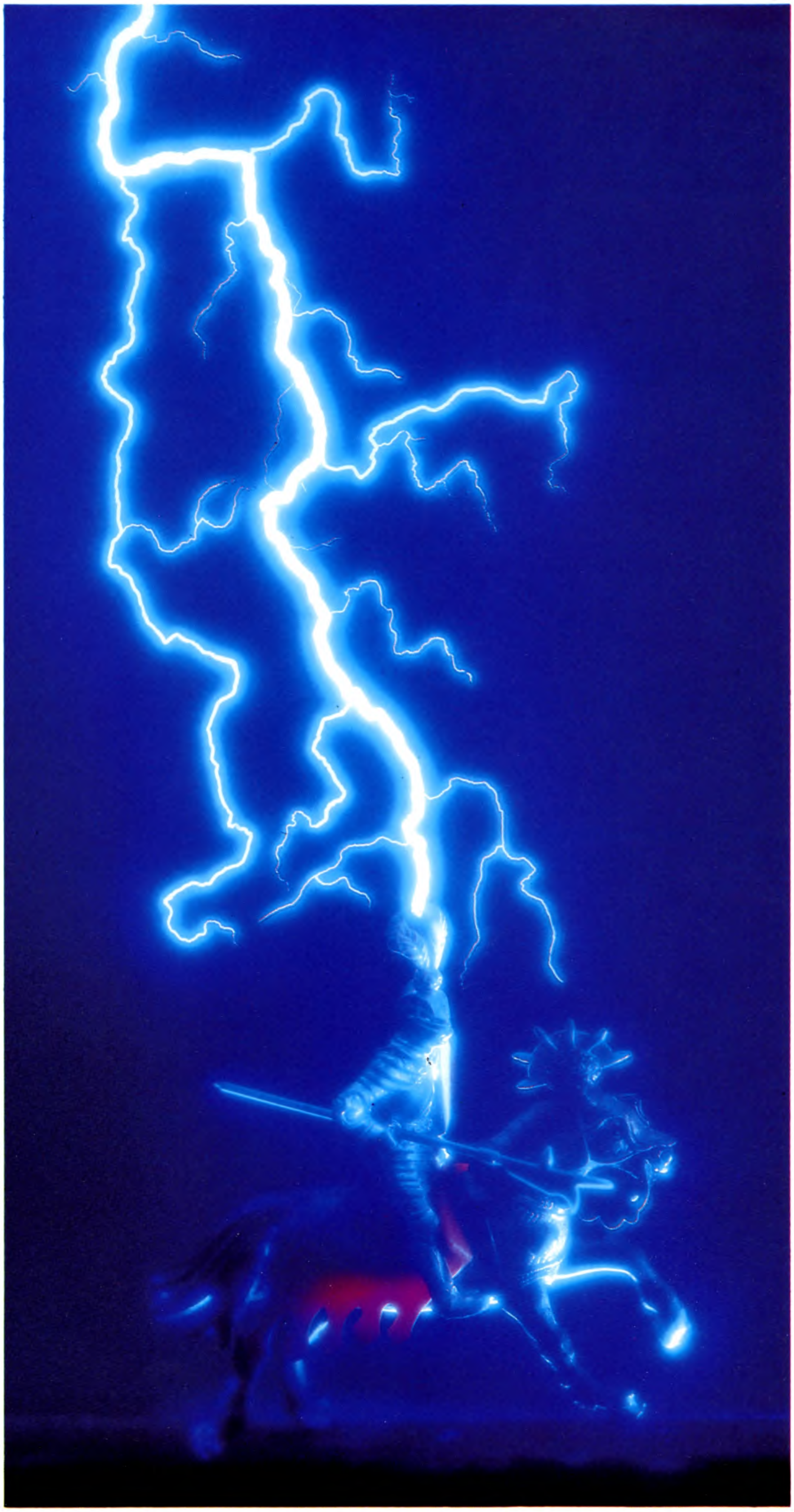
```
360 DIM t$(8,12): DIM o$(5,12): DIM
   w$(5,9): DIM m$(5,12): DIM a$(4,12):
   DIM r$(4,12): DIM c(8)
416 LET sp=1
1665 IF wn>8 THEN LET c(wn-8)=8
1666 IF lo>8 THEN LET c(lo-8)=T(wn,2)
1760 LET ra=st: LET rb=sh: LET rc=fx: LET
   rd=fy: GOSUB 3000
2140 REM Enemy selects action
2142 REM Dummy for repeat loop
2143 LET r=FN r(10)
2144 IF r=1 OR T(sp,1)>3 THEN LET
   sp=FN r(8)
2145 IF r=1 AND T(sp,1)>3 THEN GOTO
   2142
2150 IF c(e-8)=8 THEN RETURN
2155 IF c(e-8)<>0 THEN LET T(e,1)=3:
   LET T(e,2)=c(e-8): LET c(e-8)=0:
   RETURN
2170 IF T(e,3)=2 THEN LET ra=1: LET
   rb=e: LET rc=5: LET rd=5: GOSUB
   3000: IF gp<>-1 THEN LET T(e,1)=1:
   RETURN
2180 LET T(e,1)=3
2181 LET hp=5: LET vp=5: LET mv=0
2182 FOR v=1 TO 8
2183 LET zp=0: GOSUB 3100
2184 IF zp<>0 THEN GOSUB 3200
2185 NEXT v
2187 IF hp<>5 AND vp<>5 THEN
   RETURN
2188 IF mv<>0 THEN LET T(e,2)=mv:
   RETURN
2189 LET hp=T(e,8)-T(sp,8):
   LET vp=T(e,9)-T(sp,9):
   GOSUB 3200
2190 RETURN
3000 REM Target in range?
3010 LET gp=-1
3020 FOR m=ra TO (ra+7)
3030 LET xx=ABS (T(m,8)-T(rb,8)): LET
   yy=ABS (T(m,9)-T(rb,9))
3040 IF xx<rc AND yy<rd AND T(m,1)<4
   THEN LET rc=xx: LET rd=yy: LET gp=m
3050 NEXT m
3060 RETURN
3100 REM Is a weak or a strong unit nearby?
3110 IF T(v,1)>3 THEN RETURN
3120 LET xx=ABS (T(v,8)-T(e,8)): LET
   yy=ABS (T(v,9)-T(e,9))
3130 IF T(v,7)>=T(e,7) AND xx<5 AND
   yy<5 THEN LET mv=T(v,2)
3140 IF xx<hp AND yy<vp THEN LET
   hp=xx: LET vp=yy: LET zp=1
   : RETURN
3150 RETURN
3200 REM Move towards a predetermined
   square
3210 IF hp>=0 THEN LET lp=1
3220 IF hp<0 THEN LET lp=3: LET
   hp=ABS (hp)
3230 IF vp>=0 AND ABS (vp)>hp THEN
   LET lp=2
3240 IF vp<0 AND ABS (vp)>hp THEN LET
   lp=4: LET vp=ABS (vp)
3250 LET T(e,2)=lp
3260 RETURN
```

```
360 DIM T$(7),O$(4),W$(4),M$(4),A$(3),
   R$(3),CF(8)
415 W$="NNYY":S%=0
1665 IF WN>7 THEN
   CF(WN-8)=8
1666 IF LO>7 THEN
   CF(LO-8)=T(WN,1)
1760 RA=BG:RB=SH:RC=FX:RD=FY:
   GOSUB 3000
2140 REM
2142 REM
2143 R=FNR(10)
2144 IF R=1 OR T(S%,0)=4 THEN
   S%=FNR(8)-1
2145 IF R=1 OR T(S%,0)=4 THEN
   2142
2150 IF CF(E-8)=8 THENRETURN
2160 IF CF(E-8)<>0 THEN T(E,0)=2:
   T(E,1)=CF(E-8):CF(E-8)=0:RETURN
2170 IFT(E,2)=1THENRA=0:RB=E:RC=5:
   RD=5:GOSUB3000:IFGP<>-1THEN
   T(E,0)=0:RETURN
2180 T(E,0)=2
2181 H%=5:V%=5:MV=0
2182 FOR SC=0 TO 7
```

```
2183 Z%=Ø:GOSUB31ØØ
2184 IF Z%<>Ø THEN GOSUB 32ØØ
2185 NEXT SC
2187 IF H%<>5 AND V%<>5 THEN RETURN
2188 IF MV<>Ø THEN T(E,1)=MV:RETURN
2189 H%=T(E,7)-T(S%,7):V%=T(E,8)-T
    (S%,8):GOSUB 32ØØ
219Ø RETURN
3ØØØ REM
3Ø1Ø GP=-1
3Ø2Ø FOR M=RA TO (RA+7)
3Ø3Ø XX=ABS(T(M,7)-T(RB,7)):YY=ABS
    (T(M,8)-T(RB,8))
3Ø4Ø IF XX<RC AND YY<RD AND T(M,Ø)
    <4 THEN RC=XX: RD=YY:GP=M
3Ø5Ø NEXT M
3Ø6Ø RETURN
31ØØ REM
311Ø IF T(SC,Ø)=4 THEN RETURN
312Ø XX=ABS(T(SC,7)-T(E,7)):YY=ABS(T
    (SC,8)-T(E,8))
313Ø IF T(SC,6)>=T(E,6) AND XX<5 AND
    YY<5 THEN MV=T(SC,1)
314Ø IF XX<H% AND YY<V% THEN H%=
    XX:V%=YY:Z%=1:RETURN
315Ø RETURN
32ØØ REM
321Ø IF H%>=Ø THEN L%=2
322Ø IF H%<Ø THEN L%=4: H%=ABS(H%)
323Ø IF V%>=Ø AND ABS(V%)>H% THEN
    L%=1
324Ø IF V%<Ø AND ABS(V%)>H% THEN
    L%=3:V%=ABS(V%)
325Ø T(E,1)=L%
326Ø RETURN
```

Retype the original Lines 21Ø to 29Ø plus a new Line 3ØØ CHAIN"GAME". SAVE this as "WARGAME". Now LOAD your existing game. Delete Lines 21Ø to 29Ø and Lines 65 and 2Ø1Ø and then add the following lines (some of which overwrite existing lines). SAVE this after "WARGAME" as "GAME". To use the program, LOAD and RUN "WARGAME", which will then CHAIN the main program.

```
36Ø DIMtype$(7),order$(4),cf(8)
37Ø dir$="NWSE"
38Ø FORi=ØTO4:READorder$(i):NEXT
39Ø FORi=ØTO7:READtype$(i):NEXT
415 S%=Ø
1665 IF win>7 THEN cf(win-8)=8 ELSE
    cf(lose-8)=T%(win,1)
176Ø PROCrng(st,sht,fx,fy)
193Ø FORj=Ø TO 3
198Ø A%=(INSTR("FfHhMm",g$)+1)DIV2-1
214Ø DEF PROCensl
2142 REPEAT
2143 R=RND(1Ø)
2144 IFR=1 OR T%(S%,Ø)=4 THEN
```



```
   S%=RND(8)-1
2145 UNTIL R<>1 OR T%(S%,Ø)<>4
215Ø IF cf(en-8)=8 THEN ENDPROC
2155 IF cf(en-8)<>Ø THEN T%(en,Ø)=2:
   T%(en,1)=cf(en-8):cf(en-8)=Ø:
   ENDPROC
217Ø IF T%(en,2)=1 THENPROCrng
   (Ø,en,5,5):IF G%<>-1 THEN T%(en,
   Ø)=Ø:ENDPROC
218Ø T%(en,Ø)=2
2181 H%=5:V%=5:mv=Ø
2182 FORsc=Ø TO 7:Z%=Ø:PROCwk
2184 IFZ%<>Ø THEN PROCvg(en)
2185 NEXT
2187 IFH%<>5 AND V%<>5 THEN
   ENDPROC
2188 IF mv<>Ø THEN T%(en,1)=mv:PRINT
   "Retreat":ENDPROC
2189 H%=T%(en,7)-T%(S%,7):V%=T%
   (en,8)-T%(S%,8):PROCvg(en)
219Ø ENDPROC
257Ø DATAfire,halt,move,status,rout,knights,
   sergeants,men-at-arms,men-at-arms,archers,
   archers,peasants,peasants
3ØØØ DEF PROCrng(a,b,c,d)
3ØØ5 G%=-1
3Ø1Ø FOR m=a TO (a+7)
3Ø2Ø IF ABS(T%(m,7)-T%(b,7))<c AND
   ABS(T%(m,8)-T%(b,8))<d AND T%(m,
   Ø)<4 THEN c=ABS(T%(m,7)-T%(b,7)):
   d=ABS(T%(m,8)-T%(b,8)):G%=m
3Ø3Ø NEXT
3Ø4Ø ENDPROC
31ØØ DEF PROCwk
31Ø5 IF T%(sc,Ø)=4 THEN ENDPROC
311Ø IF T%(sc,6)>=T%(en,6) AND (ABS
   (T%(sc,7)-T%(en,7))<5 AND ABS
   (T%(sc,8)-T%(en,8))<5) THEN mv=T%
   (sc,1)
312Ø IF ABS(T%(sc,7)-T%(en,7))<H%
   AND ABS(T%(sc,8)-T%(en,8))<V%
   THEN H%=T%(sc,7)-T%(en,7):V%=
   T%(sc,8)-T%(en,8):Z%=1:ENDPROC
313Ø ENDPROC
32ØØ DEF PROCvg(en)
321Ø IF H%>=Ø THEN L%=2
322Ø IF H%<Ø THEN L%=4:H%=ABS(H%)
323Ø IF V%>=Ø AND ABS(V%)>H% THEN
   L%=1
324Ø IF V%<Ø AND ABS(V%)>H% THEN
   L%=3:V%=ABS(V%)
325Ø T%(en,1)=L%
326Ø ENDPROC
```

If you have a DFS then type this after SAVEing the program but before RUNning it.

```
*TAPE
FOR A%=Ø TO &198Ø:?(&EØØ+A%)=
   ?(PAGE+A%):NEXT
PAGE=&EØØ
OLD
```

Before you type in these lines, the routines starting at Line 3ØØØ need to be moved downwards to make space. Type:

```
RENUM 4ØØØ,3ØØØ
```

Then enter these lines:

```
36Ø DIM T$(8),O$(5),W$(5),M$(5),A$(4),
   R$(4),C(8)
416 SP=1
1665 IF WN>8 THEN C(WN-8)=8
1666 IF LO>8 THEN C(LO-8)=T(WN,2)
176Ø RA=ST:RB=SH:RC=FX:RD=FY:
   GOSUB3ØØØ
214Ø REM ENEMY SELECTS ACTION
2142 REM DUMMY FOR REPEAT LOOP
2143 R=RND(1Ø)
2144 IF R=1 OR T(SP,1)>3 THEN
   SP=RND(8)
2145 IF R=1 AND T(SP,1)>3 THEN 2142
215Ø IF C(E-8)=8 THEN RETURN
2155 IF C(E-8)<>Ø THEN
   T(E,1)=3:T(E,2)=C(E-8):C(E-8)=Ø
   :RETURN
217Ø IF T(E,3)=2 THEN RA=1:RB=E:
   RC=5:RD=5:GOSUB3ØØØ:IF
   GP<>-1 THEN T(E,1)=1
   :RETURN
218Ø T(E,1)=3
2181 HP=5:VP=5:MV=Ø
2182 FOR V=1 TO 8
2183 ZP=Ø:GOSUB 31ØØ
2184 IF ZP<>Ø THEN GOSUB 32ØØ
2185 NEXT V
2187 IF HP<>5 AND VP<>5 THEN
   RETURN
2188 IF MV<>Ø THEN T(E,2)=MV:
   RETURN
2189 HP=T(E,8)-T(SP,8):VP=T(E,9)-
   T(SP,9):GOSUB32ØØ
219Ø RETURN
3ØØØ REM TARGET IN RANGE?
3Ø1Ø GP=-1
3Ø2Ø FOR M=RA TO (RA+7)
3Ø3Ø XX=ABS(T(M,8)-T(RB,8)):YY=ABS
   (T(M,9)-T(RB,9))
3Ø4Ø IF XX<RC AND YY<RD AND
   T(M,1)<4 THEN
   RC=XX:RD=YY:GP=M
3Ø5Ø NEXTM
3Ø6Ø RETURN
31ØØ REM IS A WEAK OR A STRONG UNIT
   NEARBY?
311Ø IF T(V,1)>3 THEN RETURN
312Ø XX=ABS(T(V,8)-T(E,8)):YY=ABS(T
   (V,9)-T(E,9))
313Ø IF T(V,7)>=T(E,7) AND XX<5 AND
   YY<5 THEN MV=T (V,2)
314Ø IF XX<HP AND YY<VP THEN
```

```
   HP=XX:VP=YY:
   ZP=1:RETURN
3150 RETURN
3200 REM MOVE TOWARDS A
   PREDETERMINED SQUARE
3210 IF HP>=0 THEN LP=1
3220 IF HP<0 THEN LP=3:
   HP=ABS(HP)
3230 IF VP>=0 AND ABS(VP)>HP THEN
   LP=2
3240 IF VP<0 AND ABS(VP)>HP THEN
   LP=4:VP=ABS(VP)
3250 T(E,2)=LP
3260 RETURN
```

HOW IT WORKS

The additions before Line 2140 initialize some extra variables and DIMension some new arrays which will be used in the intelligence routines.

At the start of the Select Action routine there is a one in ten chance of the computer changing its target square. The target square is checked and adjusted if necessary, but the computer takes no action on it at this stage—the rest of the tests have to be carried out first.

If the unit consists of archers, Line 2170 uses the In Range routine to decide whether to fire at an enemy unit. If the archers do not fire, or the unit is of any other type, Line 2180 changes the order to move. The next section of program checks each of the enemy units. Line 2183 uses the Weak or Strong Unit subroutine to determine if the computer's unit should move towards (engage in combat) or move away from the nearest unit belonging to the player.

If the unit controlled by the computer isn't engaged in combat, or about to do so, the program uses the Move Towards subroutine (at Line 3500) to move the unit towards the target square.

YOUR CHOICE

You can incorporate either or both of the extra strength or intelligence routines to improve the computer's play at Cavendish Field—according to the strength of opponent you feel competent to take on.

The very nature of heuristics means that the rules of thumb that have been incorporated into the program aren't necessarily the best in all circumstances. And some players will find these tactics easier to outwit than others.

Only trial and error can suggest the best compromises and you may wish to try some different routines—you may well find you can further strengthen the computer's play with a little effort.

CLIFFHANGER: THE HIGH JUMP

Willie is beginning to get a little annoyed with his lot. He dislikes boulders being rolled down the slope at him so he starts to jump up and down on the spot

In part two of this three part article, Willie is going to start moving about a bit more. This time you are going to concentrate on giving him an up-and-down action so that he can leap over oncoming boulders.

Last time you taught Willie how to walk, but this won't help when the boulders start rolling. So this time you are going to make him jump in the air so that the boulder can roll past, underneath him.

```
     org 59336
jmp  ld de,6
     ld hl,759
     call 949
     ld a,(57335)
     cp 1
     jr nz,mjb
     inc a
     ld (57335),a
     ld hl,(57332)
     ld de,32
     sbc hl,de
     ld (57332),hl
     ld bc,57048
     ld a,40
     ld de,259
     call 58970
     ret
mjb  cp 2
     jr nz,mjc
     inc a
     ld (57335),a
     ld hl,(57332)
     ld bc,57000
     ld a,40
     ld de,258
     call 58970
     ld de,64
     add hl,de
     ld a,45
     ld bc,15616
     call 58217
     ret
mjc  cp 3
     jr nz,mjd
     inc a
     ld (57335),a
     ld hl,(57332)
     ld bc,57048
     ld a,40
     ld de,259
     call 58970
     ld de,32
     add hl,de
     ld (57332),hl
     ret
mjd  cp 4
     jr nz,mfj
     ld a,0
     ld (57335),a
     ld hl,(57332)
     ld de,32
     sbc hl,de
     ld bc,15616
     ld a,45
     call 58217
     jp 59153
mfj  ret
```

This routine butts straight onto the end of the last routine, overwriting the temporary **ret** that was put there. It begins with the label **jmp**, meaning jump, so the first thing to be done is to play the jumping tune.

This is done by calling the BEEP routine in ROM at 949 and feeding the appropriate parameters to it in the DE and HL registers. For the significance of these parameters see page 732.

JUMPED UP

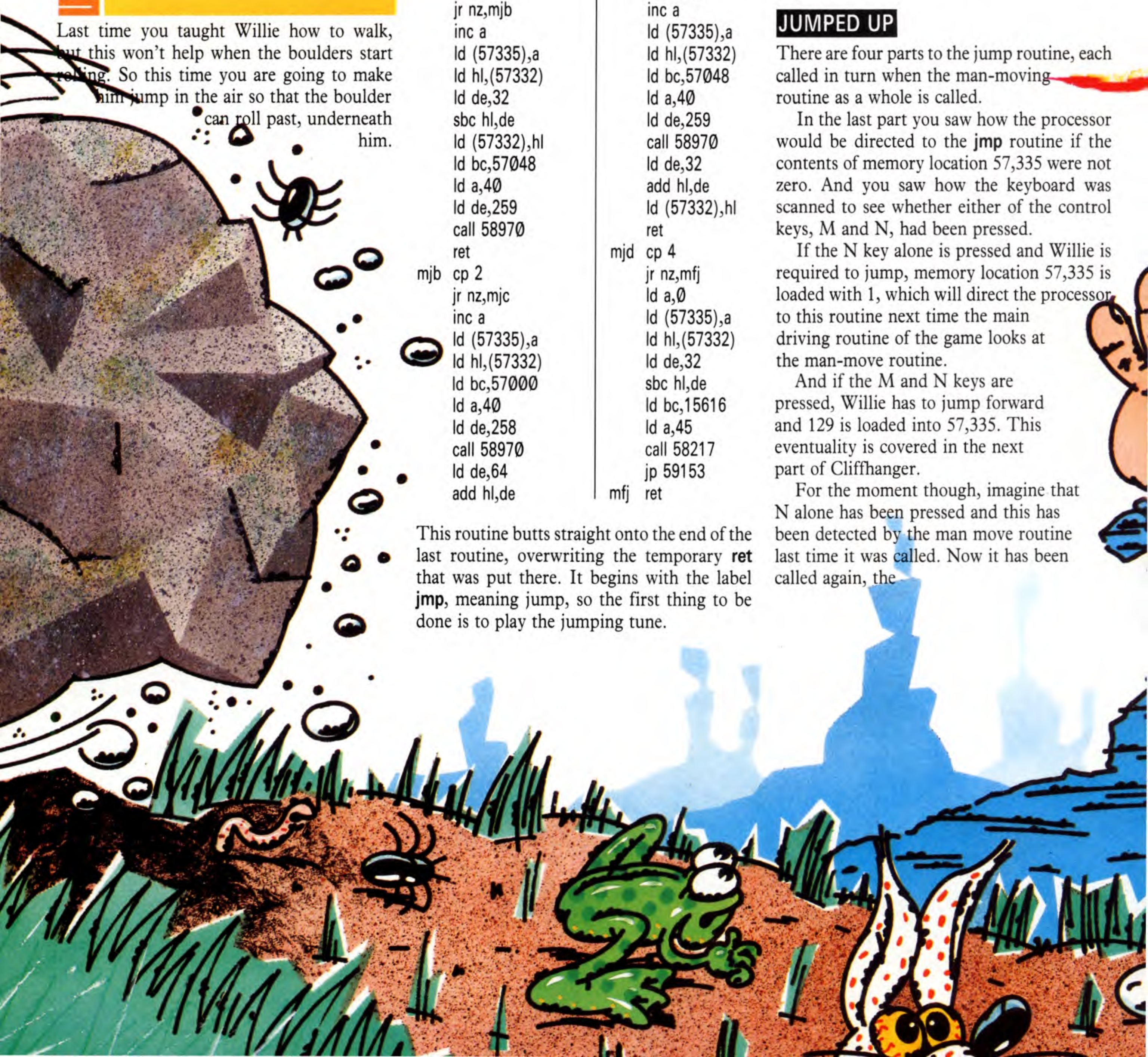
There are four parts to the jump routine, each called in turn when the man-moving routine as a whole is called.

In the last part you saw how the processor would be directed to the **jmp** routine if the contents of memory location 57,335 were not zero. And you saw how the keyboard was scanned to see whether either of the control keys, M and N, had been pressed.

If the N key alone is pressed and Willie is required to jump, memory location 57,335 is loaded with 1, which will direct the processor to this routine next time the main driving routine of the game looks at the man-move routine.

And if the M and N keys are pressed, Willie has to jump forward and 129 is loaded into 57,335. This eventuality is covered in the next part of Cliffhanger.

For the moment though, imagine that N alone has been pressed and this has been detected by the man move routine last time it was called. Now it has been called again, the



- BLANKING OUT WILLIE'S FEET
- SECTIONALIZING THE JUMP
- PRODUCING THE JUMP
- SOUND EFFECTS
- ADJUSTING SCREEN POINTERS

The 'CLIFFHANGER' listings published in this magazine and subsequent parts bear absolutely no resemblance to, and are in no way associated with, the computer game called 'CLIFF HANGER' released for the Commodore 64 and published by New Generation Software Limited.

processor has been directed straight to the routine given in this part of Cliffhanger and the tune has been played. Now read on....

The contents of memory location 57,335 are loaded up into the accumulator again. Although they were already in the accumulator before this routine was called it is more than likely that the **BEEP** routine has used the accumulator in the course of making the jumping sound effect, so it is best to be on the safe side and load them into the accumulator again.

The contents of A are then compared to 1. Naturally 1 is found, so the **jr nz,mjb** instruction following does not operate and the processor goes straight on to **inc a**. This increments the contents of the accumulator—to 2—then the 2 is loaded back into 57,335 by **ld (57335),a**. This increments the jump counter so that next time the man move routine is called it will jump straight to the jmp routine, but when it gets there it will skip this first part of the jump and go straight on with part two. As you can see, after it had hit the **cp 1** instruction the following **jr nz,mjb** instruction would jump the processor forward to the second part.

Willie's screen position is then loaded into the HL register from memory location 57,332. 32 is subtracted from it by loading the 32 into the DE and performing a **sbc hl,de**. This effectively moves the pointer to Willie's position one character square up on the screen. The result is loaded back into the memory location that carries Willie's screen position, 57,335.

Even though this new screen position has been loaded back it still remains in the HL register. All **lds** are essentially copying operations which leave the value loaded both in the new location or register it is loaded into, and the old location or register it is loaded from. This is important here, as you are about to call the block print routine at 58,97Ø and you need the screen position where you are going to print in HL.

BC is loaded with the data pointer to the first figure of the man jumping at 57,Ø48. This data was given in an earlier part of Cliffhanger and was read into a data table in memory.

A is loaded with 4Ø—which is the attribute for blue on cyan, Willie's colour. And DE is set 259. This tells the block print routine to print a block one by three—259 loads 1 into the D register and 3 into the E register, 1 × 256 + 3 = 259. Then, the block print routine at 58,97Ø is called. This prints a picture of Willie jumping.

The processor then returns.

GOING UP

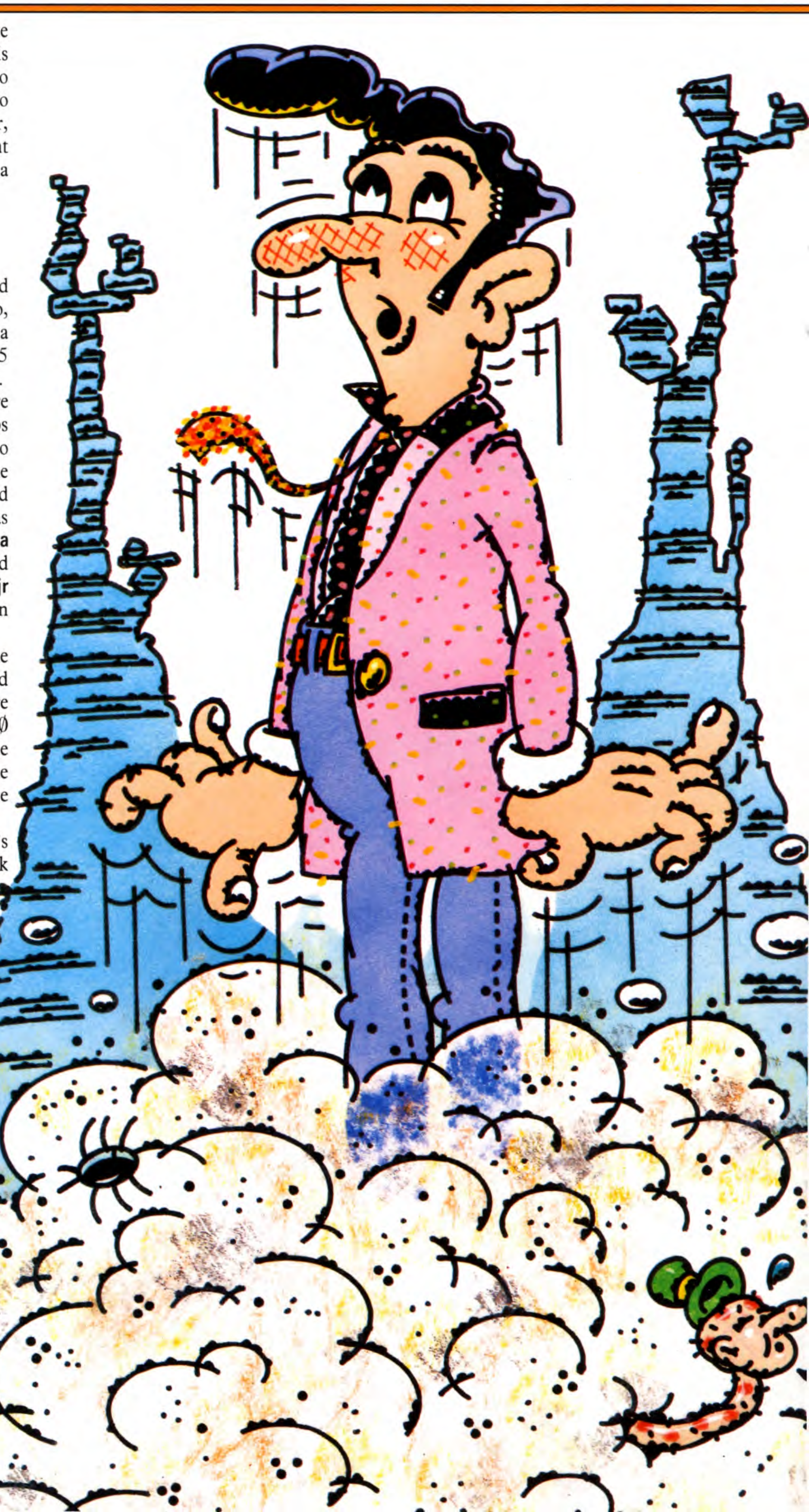
The next part of the jump routine is called next time the man move routine is referred to, as you have seen. Again, this starts off with a compare to see if memory location 57,335 contains 2—which, the second time, it does.

Remember that the contents of 57,335 are in the accumulator when the processor jumps to this part of the routine, so you don't have to load it up again. If the 2 is found, the processor continues with this routine and increments the jump counter again as instructed by the **inc a** and **ld (57335),a** instructions. So when the routine is called next time, the **cp 2** will not find a 2 and the **jr nz,mjb** instruction will send the processor on to part three.

If the 2 was found and this part of the routine is required, Willie's new, elevated position is loaded into HL from its storage location in 57,332. BC is loaded with 57,ØØØ which is the address of the data for Willie standing with his legs together. This is the picture that is going to be printed at the height of Willie's leap.

A is loaded with 4Ø—blue on cyan, Willie's colour. And DE is set to 258 to print a block of one by two—1 × 256 + 2 = 258. The block print routine at 58,79Ø prints Willie with his legs together up in the air.

DE is loaded with 64 and added to the screen pointer in HL, which moves two character squares down the screen. HL points to Willie's head, remember, so subtracting 64 (32 for each row of squares) makes it point to the square below Willie's feet. A is loaded with 45—cyan on cyan, just a regular piece of sky. And BC, the data pointer, is loaded with 15,616, the beginning of the character set which is a blank space.



The character print routine at 58,2Ø8 is called, which prints one blank space to blot out the bit of Willie left from the first part of the jump routine. Remember, a one-by-three block was printed on the screen there, and only a one by two block in part two. The processor then returns again, until next time.

JUMP DOWN

Next time, the processor skips straight to the third part of the jump routine. Firstly, it checks whether it needs to go on to the fourth part of the jump routine or the section given in the next instalment of Cliffhanger which allows Willie to jump forward.

Then it decrements the jump counter which is still in the accumulator and stores it back in 57,335 so that the processor will jump on to the fourth part of the jump routine next time.

Willie's screen pointer is loaded back up into the HL register. The data pointer in BC is set back to the beginning of the same picture data that was used in part one of the jump routine. Willie comes down looking the same as he did going up.

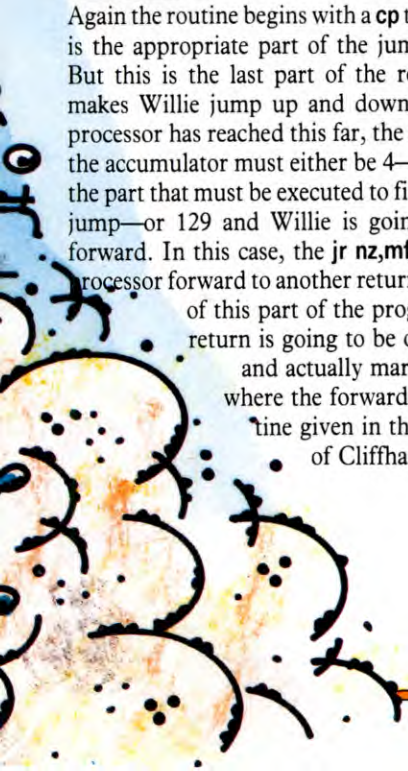
A is loaded with 4Ø again—blue on cyan. And DE is set with 259 again—one by three. The block print routine at 58,97Ø is called again. This prints up Willie descending.

DE is loaded with 32 which is then added to HL. The result is loaded back into 57,332. This moves the screen pointer down one character square.

The processor then returns, with the data locations set up for the next part.

JUMP OFF

Again the routine begins with a **cp** to see if this is the appropriate part of the jump routine. But this is the last part of the routine that makes Willie jump up and down. So if the processor has reached this far, the contents of the accumulator must either be 4—and this is the part that must be executed to finish off the jump—or 129 and Willie is going to jump forward. In this case, the **jr nz,mfj** sends the processor forward to another return at the end of this part of the program. This return is going to be overwritten and actually marks the spot where the forward jump routine given in the next part of Cliffhanger starts.



This time, though, once the test has been passed, there is no need to increment the jump counter. Instead it has to be reset to Ø so that next time the man move routine is called the processor executes the walk routine given in the last part of Cliffhanger and checks once more to see if any of the keys have been pressed. This is done by the **ld a,0** and the **ld (57335),a.**

Willie's screen position is loaded from 57,332 into the HL register again. Then 32 is subtracted from it via the DE register. This moves the pointer up the screen one character square.

BC is loaded with the address of the space, A is loaded with the attribute for cyan on cyan and the print routine at 58,2Ø8 is called again. This overprints the extra character square occupied by Willie that was printed up by stage three of this jump routine.

The **jp 59153** instruction takes the processor back again, to the beginning of the man move routine given in the last part of Cliffhanger. As the jump counter in 57,335 is now set to Ø, the processor then goes on to the routine which overprints the rest of Willie given there.

The processor finds its return in that program. The **ret** at the end of this routine is simply to prevent the program crashing if the jump forward routine is called. It will be overwritten by the next part of Cliffhanger.

In this part of the program you make Willie move up and down and make him jump forward. Again these are small separate routines that are going to be called by the main man-moving routine given in the next part of Cliffhanger.

MOVE ON UP

This little routine moves Willie up:

```
ORG  21584
DEC  $CØ11
LDA  $DØØ1
SEC
SBC   #4
STA  $DØØ1
RTS
```

As always, you have two sets of coordinates to deal with when moving something about on the screen. First of all, Willie's double density Y coordinate in memory location $CØ11 has to be decremented. This moves it half a character square up the screen.

Next the Y coordinate of sprite Ø—the sprite that carries Willie's picture—is loaded into the accumulator. The carry flag is then set by the SEC instruction. A subtraction is about to be done by the processor.

The SBC #4 then subtracts 4 from the Y coordinate of Willie's sprite—this moves it four pixels, or half a character square, up the screen—and the STA stores it back into the memory location $DØØ1 which carries the Y coordinate of sprite Ø.

The processor then returns.

GET DOWN

The following routine moves Willie down the screen four pixels, or half a character square:

```
ORG  215Ø4
INC  $CØ11
LDA  $DØØ1
CLC
ADC   #4
STA  $DØØ1
RTS
```

This works in almost exactly the same way as the routine to move Willie up. First the double density Y coordinate in $CØ11 is incremented instead of decremented to move him down.

The Y coordinate of sprite Ø is loaded up into the accumulator then the carry flag is cleared. This time an addition is going to be done.

The ADC #4 adds 4 to Willie's Y coordinate, which moves him four pixels down, and the result is stored back in $DØØ1.

UP RIGHT

The next routine moves Willie up and to the right so that he can clear an incline or jump over a pothole or snake.

```
ORG  21248          |   STA  $DØØØ
LDA  $DØØ1          |   LDA   #Ø
SEC                 |   ROL  A
SBC   #4            |   STA  $Ø384
STA  $DØØ1          |   LDA  $DØ1Ø
DEC  $CØ11          |   EOR  $Ø384
LDA  $DØØØ          |   STA  $DØ1Ø
CLC                 |   INC  $CØ12
ADC   #4            |   RTS
```

This routine starts off like the one that moves Willie up. It loads the Y coordinate of Willie's sprite into the accumulator, sets the carry flag, subtracts 4 and stores it back in $DØØ1.

Then the double density Y coordinate is decremented. That's the up part taken care of. Now to move the sprite to the right.

The X coordinate of sprite Ø, which is stored on $DØØØ, is loaded into the accumulator. The carry flag is cleared again and 4 is added to the X coordinate of Willie's sprite. The result is stored back in $DØØØ.

The accumulator is then loaded with Ø and a ROtate Left command is used to move any overflow from the last addition. The result is stored temporarily in a convenient location in the tape buffer, memory location $Ø384. If the X coordinate of the sprite has been incremented over 256, the appropriate bit of the MSB register in memory location $DØ1Ø must be set.

As this is sprite Ø, bit zero of the MSB register is the one that has to be set. So the contents of the MSB register are loaded into the accumulator and exclusively ored with the contents of $Ø384—which either contains Ø, if the X coordinate was not incremented over 256, or 1 if it was. The result is stored back in $DØ1Ø.

The double density X coordinate also has to be incremented. This is done by the INC $CØ12 instruction. The processor then returns to the main man-moving routine which will be given in the next part of Cliffhanger.

This program tests the keys and applies gravity to Willie—it makes him drop down if he is not standing on firm ground. Don't forget to set your computer up as normal before you key in this program.

```
3Ø FORPASS=ØTO3STEP3
4Ø P%=&1EEE
5Ø [OPTPASS
6Ø .Keys
7Ø LDA&7C
8Ø AND#1
9Ø BEQLb1
1ØØ RTS
11Ø .Lb1
12Ø LDA&7C
13Ø AND#&7F
14Ø STA&7C
15Ø LDA#Ø
16Ø LDX&7A
17Ø LDY&7B
18Ø DEY
19Ø DEY
2ØØ JSR&1DBD
21Ø CMP#15
22Ø BEQLb2
23Ø CMP#16
24Ø BEQLb2
25Ø CMP#17
26Ø BEQLb2
27Ø LDA&7C
28Ø AND#&18
29Ø LSRA
3ØØ ORA#&1Ø
31Ø ORA&7C
32Ø STA&7C
33Ø RTS
34Ø .Lb2
35Ø LDA&7C
36Ø AND#&7
37Ø STA&7C
38Ø LDA#&81
39Ø LDY#&FF
4ØØ LDX#&FF
41Ø JSR&FFF4
42Ø TXA
43Ø BEQLb3
44Ø LDA&7C
45Ø ORA#&8Ø
46Ø STA&7C
465 JSR&1BEB
47Ø .Lb3
48Ø LDA#&81
49Ø LDY#&FF
5ØØ LDX#&AA
51Ø JSR&FFF4
52Ø TXA
53Ø BEQLb4
54Ø LDA&7C
55Ø ORA#&4Ø
56Ø STA&7C
57Ø .Lb4
58Ø LDA#&81
59Ø LDY#&FF
6ØØ LDX#&9A
61Ø JSR&FFF4
62Ø TXA
63Ø BEQLb5
64Ø LDA&7C
65Ø ORA#&2Ø
66Ø STA&7C
67Ø .Lb5
68Ø LDA#&81
69Ø LDY#&FF
7ØØ LDX#&EF
71Ø JSR&FFF4
72Ø TXA
73Ø BEQLb6
74Ø LDA&8Ø
75Ø ORA#&8Ø
76Ø STA&8Ø
77Ø .Lb6
78Ø LDA#&81
79Ø LDY#&FF
8ØØ LDX#&AE
81Ø JSR&FFF4
82Ø TXA
83Ø BEQLb7
84Ø LDA&8Ø
85Ø AND#&7F
86Ø STA&8Ø
87Ø .Lb7
88Ø RTS
89Ø ]NEXT
```

Warning: do not attempt to CALL this program. It will not work without the routine in the next part of Cliffhanger.

IS WILLIE SQUARE?

The contents of Willie's status location at &7C is loaded into the accumulator. The bits of this location tell Willie what to do next. The meaning of each bit is explained in the last part of Cliffhanger on page 1342.

Here the contents of this location are ANDed with 1 to check whether Willie is in a half character position. If he is, bit zero is set and the BEQ instruction in Line 9Ø does not make the processor branch over the RTS in Line 1ØØ.

Willie can only stop in a full character position. So if he is in the half character position he is still in mid-move and this routine need not be executed.

LOOK BELOW YOU!

Willie's jump status is loaded up from &7C again. If Willie jumped last time, the jump bit has to be switched off this time round. So the status byte is ANDed with &7F which clears bit seven and the result is stored back in &7C by the instruction in Line 14Ø. The accumulator is then cleared by loading the number Ø into it.

The X and Y coordinates of Willie are then loaded up into the X and Y registers from &7A and &7B again. And Y is decremented twice to point to the character below Willie's feet. The processor then jumps to the subroutine at &1DBD to look at the character below Willie.

The result is returned in the accumulator and compared with 15, 16 and 17—the UDG numbers for a level piece of ground, a piece of ground sloping up to the right and a piece of ground sloping down to the right respectively.

If ground of any description is found, the BEQ instructions in Lines 22Ø, 24Ø and 26Ø branch the processor forward to the label Lb2 in Line 34Ø. If ground is not found, Willie is either up in the air, in mid-jump, or he has fallen down a hole, and the processor continues.

DECLINE AND FALL

If Willie is already dead—that is, he is falling three character squares and bit three of &7C is set—there is nothing you can do for him. But if he is in mid-leap, he must be brought down one extra character square. So his status in &7C is loaded up into the accumulator and ANDed with &18. This isolates bits four and five. The contents of these two bits are then logically shifted to the right by the LSRA in Line 29Ø.

This means that if bit five was set *and* Willie was in mid-jump *and* was to have fallen one character square, bit four is now set and he has to fall two character squares, perhaps because he has gone over the edge of the slope. And if bit four was set and he had to fall two character squares, bit three is now set, he is dead so he has to descend three character squares because he has fallen down a hole.

If none of them are set, Willie has to drop down one character square at least, so bit five is set by ORing the result with &1Ø.

The result of all that is ORed with &7C—so that the other bits of the status byte are preserved but the newly set fall bits are added—and the result is stored back in &7C by the instruction in Line 32Ø.

The processor then hits the RTS in Line 33Ø and leaves the routine.

THE KEY TO MOVEMENT

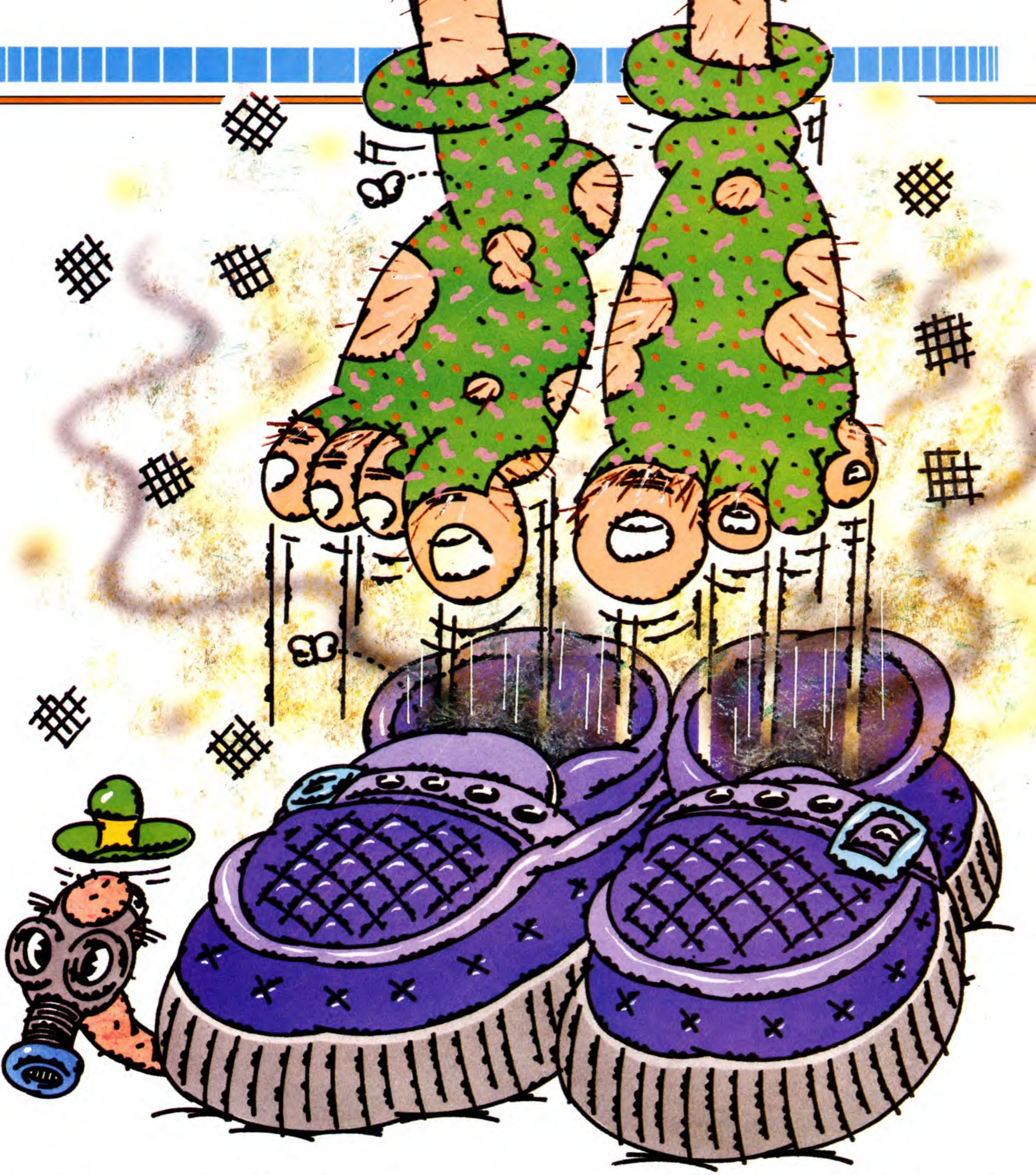
The processor is about to look at the keyboard to see what Willie is supposed to do next. But first it stops him in his tracks until it finds out where to move him.

The contents of &7C are loaded up into the accumulator, ANDed with &7 and the result is stored back in &7C. This clears the four most significant bits which control the jump and Willie's right and left movements, and his drop down one character square.

Next A is loaded with &81, Y with &FF, X with &FF and the operating system routine at &FFF4 is jumped to. This is the equivalent of an OSBYTE &81—it scans the keyboard.

But with &FF in the Y register when you enter this routine you get the equivalent of an INKEY instruction with a negative value. This scans the keyboard to see if a specific key has been pressed. The key press it is looking for is specified by the value in the X register. Here &FF specifies the SHIFT key. In Cliffhanger, if the SHIFT key is pressed, Willie jumps.

If the key specified is pressed X contains &FF when the processor comes out of the routine. If not, X contains Ø.



Here, after the processor gets back from the routine at &FFF4, the TXA transfers the result in the X register into A. If the SHIFT key has not been pressed, the BEQ instruction in Line 43Ø branches the processor forward to the label Lb3 in Line 47Ø, where the next key press is checked for.

But if the key press is made, the BEQ instruction does not operate and the processor proceeds. The contents of &7C are loaded up again, ORed with &8Ø and the result is stored back in &7C. This sets bit seven which specifies a jump, leaving the other bits alone.

The processor then jumps to the subroutine at &1BEB which gives the jump sound.

KEY LEFT GO

The instructions in Lines 48Ø to 51Ø scan the keyboard again in exactly the same way, only this time X is loaded with &AA which specifies the N key. In Cliffhanger, if the N key is pressed, Willie goes to the left.

Again the result of the scan is transferred into the accumulator and the BEQ instruction takes the processor onto the next keypress check if the N key hasn't been pressed.

But if it has, the contents of &7C are ORed with &4Ø. This sets bit six which tells Willie to go left.

THE RIGHT KEY

The instructions in Lines 58Ø to 81Ø scan the keyboard again in exactly the same way, only this time X is loaded with &9A which specifies the M key. In Cliffhanger if the M key is pressed, Willie goes to the right.

Again, the result of the scan is transferred into the accumulator and the BEQ instruction takes the processor onto the next keypress check if the M key hasn't been pressed.

But if it has, the contents of &7C are ORed with &2Ø. This sets bit five which tells Willie to go right.

Q FOR NO SONG

The instructions in Lines 68Ø to 71Ø scan the keyboard again in exactly the same way, only this time X is loaded with &EF which specifies the Q key. In Cliffhanger if the Q key is pressed, the sound is turned off.

Again, the result of the scan is transferred into the accumulator and the BEQ instruction takes the processor onto the next keypress check if the Q key hasn't been pressed.

But if it has, the contents of &8Ø are ORed with &8Ø. This sets bit seven which tells the processor that no sound is required.

SOUNDS SIBILANT

The instructions in Lines 78Ø to 81Ø scan the keyboard again in exactly the same way, only this time X is loaded with &AE which specifies the S key. In Cliffhanger, if the S key is pressed, the sound is turned back on again.

Again the result of the scan is transferred into the accumulator and the BEQ instruction takes the processor onto the RTS at the end of the routine if the S key hasn't been pressed.

But if it has, the contents of &8Ø are ANDed with &7F. This clears bit seven which tells the processor that sound is required again.

The processor then hits the RTS in Line 88Ø and returns.

Last time you taught Willie how to walk. That in itself is a significant achievement, but it won't help Willie when the boulders start rolling. So this time you are going to make him jump in the air so that the boulder can roll by underneath him.

```
       ORG 2Ø14Ø
JUM    JSR CLICK
       LDA 18261
       CMPA #1
       BNE MJA
       INC 18261
       LDX 18249
       PSHS X
       LEAX 256,X
       LDU #1536
       JSR CHARPR
       PULS X
       LEAX -256,X
       STX 18249
       LDU #17814
       JSR CHARPR
       LEAX 254,X
       LDU #17846
       JSR CHARPR
       RTS
MJA    CMPA #2
       BNE MJB
       INC 18261
       LDX 18249
       LEAX -256,X
       LDU #1787Ø
       JSR CHARPR
       LEAX 254,X
       JSR CHARPR
       LEAX 254,X
       JSR CHARPR
       RTS
MJB    CMPA #3
       BNE MJC
       INC 18261
       LDX 18249
       LEAX -256,X
       LDU #1536
       JSR CHARPR
       LEAX 254,X
       LDU #17926
       JSR CHARPR
       LEAX 254,X
       LDU #17958
       JSR CHARPR
       LEAX 254,X
       LDU #1799Ø
       JSR CHARPR
       RTS
MJC    CMPA #4
       BNE MFJ
       CLR 18261
       LDX 18249
       PSHS X
       LDU #1536
       JSR CHARPR
       PULS X
       LEAX 256,X
       STX 18249
       LDU #17774
       JSR CHARPR
       LEAX 254,X
       JSR CHARPR
       RTS
MFJ    RTS
CHARPR EQU 194Ø2
CLICK  EQU 2Ø847
```

The first thing the JUM—or jump—routine does is to jump to the subroutine CLICK. This plays one note of the jumping sound effect. But you don't have that routine yet so add a return at 2Ø847 by POKEing it with 57, so that the program does not crash when you test it.

JUMPING BY NUMBERS . . . BEGIN!

The accumulator is then loaded with the contents of memory location 18,261 which are the man-jump variable. Initially this is loaded with 1 or 129, depending on which keys are pressed and whether a vertical jump or a forward jump is required. But these jumps each come in four parts and each part is executed in turn as the man-move routine as a whole is called successively. And to achieve this, the man-jump variable is incremented as each part is performed.

If the man-jump variable is Ø, the part of the routine given last time which examines the keyboard is used to see if a jump is required. If the man-jump variable is anything other than Ø, the processor skips that part and performs one of the jump routines.

If it is 1, the first part of this routine is performed. And during the course of the routine, the man-jump variable is incremented so that next time the man-moving routine is called, this part is skipped too, and the processor jumps to part two here. Then each time the man-move routine is called it moves onto the next part given here—until part four is done and the man-jump variable is reset once again to Ø.

If the man-jump routine was set to 129 by the routine given last time, it will skip over this part of the program and perform the forward-jump routine given in the listing in the next part of Cliffhanger.

ONE!

The contents of the accumulator are compared with 1. If 1 is not found the processor skips on to MJA. But if 1 is found, the processor continues with this part of the routine.

The first thing that is done is to increment the man-jump variable in 18,261 so that it is ready for next time.

X is then loaded with Willie's position from 18,249 which is stored temporarily by pushing it onto the hardware stack. The pointer in X is incremented by 256 to move it down one character square. And U is loaded with 1536, the address of the top left-hand corner of the screen. The processor then jumps to the CHARPR subroutine which prints a block of sky over the bottom half of Willie.

As he is going to jump up in the air you need to print some sky under him. Otherwise his legs will be left behind when he jumps.

Willie's position is pulled off the stack again and moved one character square up the screen by subtracting 256. This is stored back in the position location at 18,249. U is loaded with 17,814 which is the start address for the data for the picture of Willie with his legs apart. Then CHARPR prints up his top half.

The screen pointer is then incremented by 254 to point to the start of the bottom half of Willie and U is reloaded. CHARPR is then called to print up his bottom half.

So Willie is printed up with his legs apart one character square above the ground which makes it look like he is jumping. That done, the processor returns.

TWO!

The next time the man-moving routine is called, the number in location 18,261 will be 2. So the processor will skip the routine given in the last part of Cliffhanger and the first part of the routine given above. But when it branches forward again to MJA, the contents of 18,261, which are still in the accumulator, will be compared to 2.

On subsequent calls of the man-moving routine, when the contents of 18,261 have been incremented again, the BNE MJB instruction will send the processor forward again to check for 3 or 4, or 129 or 13Ø or 131 or 132. This time though, it will perform the next little routine and print up Willie in the second stage of flight.

Again the routine starts off by incrementing the contents of 18,261, ready for next time the man-moving routine is called. Then Willie's position is loaded from 18,249 into X and decremented by 256 to move it one more character square up the screen. This time, though, the result is not stored back in 18,261. Willie is at the top of his leap.

U is loaded with 17,87Ø, the beginning of some new picture data for Willie. This time, though, it is three character squares deep. So Willie has to be printed in three sections—CHARPR has to be called three times and X decremented by 254 to move one line down the screen between each call.

That done the processor returns again.

THREE!

Next time the man-moving routine is called, location 18,261 has three in it and the processor performs the MJB routine. Again it starts off by incrementing 18,261 and loading 18,249 into X and incrementing it by 256.

This time, though, Willie has passed his zenith and is dropping again so the top part of his last picture has to be blanked out. U is loaded with 1536, the address of the top left-hand corner of the screen and when CHARPR is called it prints a piece of plain sky over what was Willie's head.

But Willie does not remain headless for long. X is incremented by 254 to move the pointer down the screen to Willie's new head position. U is loaded with 17,926, the start address of Willie's new head data. And CHARPR is called to print Willie's new head up on the screen.

The X pointer is then incremented to move it down the screen and U is loaded with the new data address to print up the rest of Willie.

FOUR!

If Willie is jumping vertically in the air, sooner or later the processor will reach MJC. The CMPA #4 and BNE condition will only send it on to MJF if Willie is jumping forward. Otherwise the last little routine here will be performed.

And as this is the last routine involved in making Willie jump vertically, the contents of location 18,261 are not incremented. CLR 18261 sets them back to Ø so that the main man-moving routine given in the last part of Cliffhanger will be performed over again, and the keyboard will be scanned to see if Willie is required in jump some more.

Willie's screen position is loaded up into X and pushed onto the hardware stack for temporary storage. As Willie is only two character squares tall when standing or walking and three character squares tall when jumping, the top square has to be blanked out again. (Note that Willie does not actually get any taller on the screen when he jumps. It is just that he spills over into three squares, using only half of the top and bottom squares to give him a smoother, half-a-square-at-a-time jump.)

U is loaded with 1536 so that it points to blank sky at the top of the screen again. And when CHARPR is called, it prints blank sky over Willie's old head.

Willie's position is then pulled off the stack and incremented by 256 to move it one character square down the screen. His position was decremented by 256 once at the beginning of this routine when he started his jump, remember. So when the new value is stored back in 18,261, it simply puts Willie's position back to what it was to start with.

U is then loaded with 17,774, the start address of the data for the picture of Willie with his legs together. CHAPR is called twice with X incremented by 254 in between to print up the whole of the picture of Willie.

PATTERNS FOR PASCAL PROGRAMS

Now that you have come to grips with the structured nature of Pascal programming, take a look at how the structures are built up and how a program evolves

The first part of this two-part article on Pascal showed how you need to work out the solution to a problem before you start to program the computer. And you need to refine your solution until it approximates to a Pascal program that can be given in terms your computer will be able to understand.

So far, so good. But you cannot hope to write a Pascal program in this way unless you know to start with what you are aiming towards. In other words, you need to know what commands and structures are supported by Pascal, so that you can plan your program in terms of these.

In fact, although it is not so obvious, this is the same process you use when you write in BASIC. Your experience of using the machine tells you what commands are available, and how you can plan what you want to do so that it works within the capability of the machine. But because BASIC does not enforce a structured approach, BASIC programmers are less conscious of this process.

So before going any further into writing programs in Pascal, the next thing to do is to look at how the structures are built up.

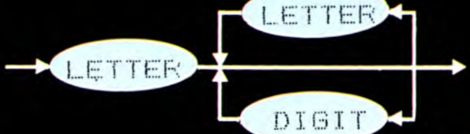
SET PIECES

In BASIC programming there are several set procedures for dealing with a variety of problems. Among these, common examples include the ability to repeat an operation a set number of times, using a FOR . . NEXT loop, or to make a decision using IF. . .THEN. Similar structures exist in Pascal, although these often allow a higher level of refinement than in BASIC. This article covers some of the most useful of these standard forms.

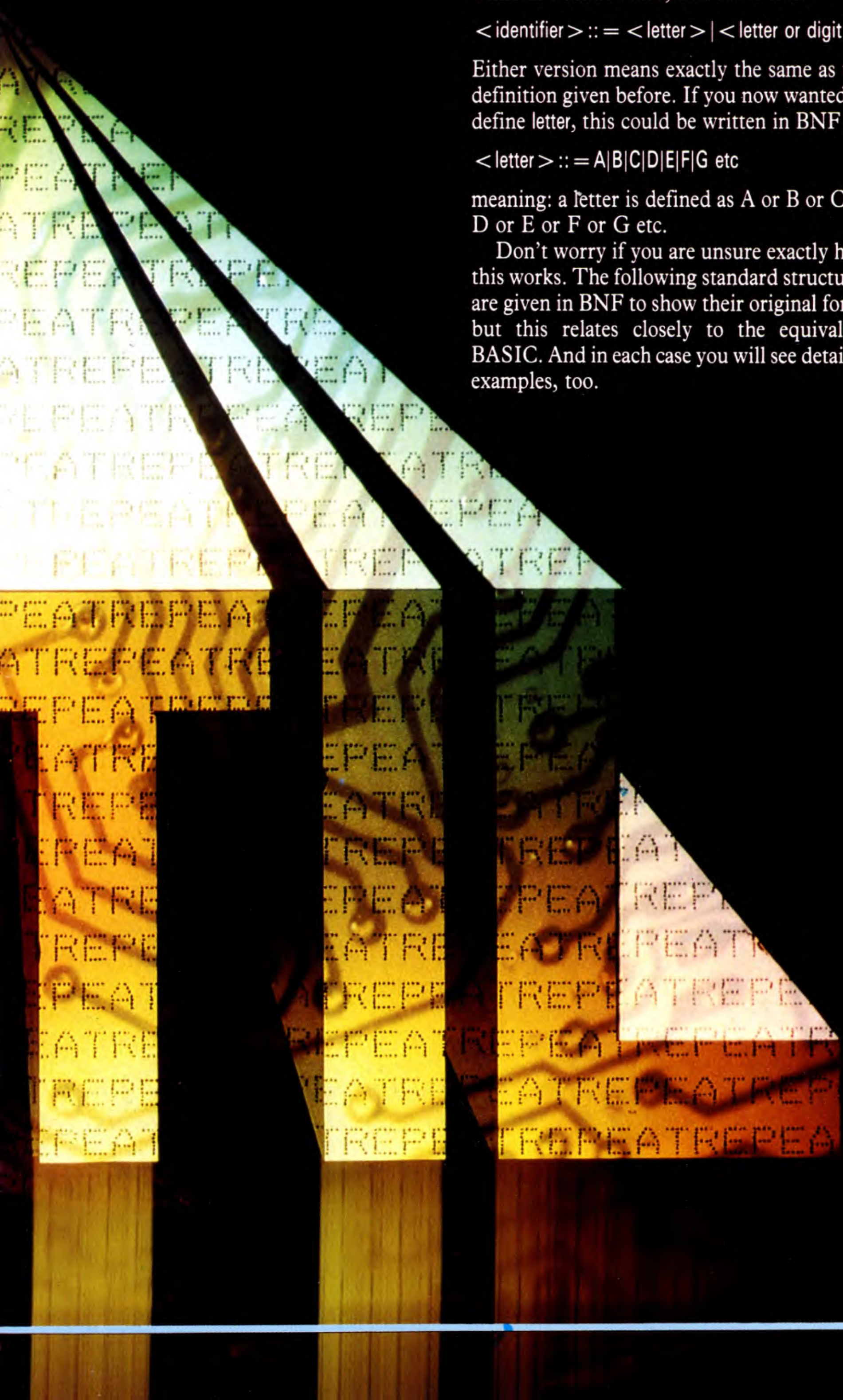
Unlike BASIC, Pascal is precisely defined, without the individual variation between different systems that mean a BBC BASIC program will not run on a Commodore, for example. This is why it is possible for the programs given later to run on any machine whose Pascal compiler accepts lower case.

When you come to write a Pascal program, this means that you must work within a precisley defined form for each statement. These standard definitions are given in a formal notation developed during the 1960s and known as Backus Naur form (BNF). They can also be given as syntax diagrams.

Either of these forms of notation simply give the generalized structure of the statement. For example, you may want to define an identifier such that: *an identifier is defined as a letter or a digit*. In syntax diagram form this is given as:



- SET PIECES
- REPETITION TECHNIQUES
- WHILE...DO
- REPEAT...UNTIL
- FOR
- DECISION MAKING
- IF...THEN...ELSE
- CASE
- DEVELOPING A PROGRAM
- ENTERING PASCAL



while in Backus Naur, this looks like:

<identifier> :: = <letter> | <letter or digit>

Either version means exactly the same as the definition given before. If you now wanted to define letter, this could be written in BNF as:

<letter> :: = A|B|C|D|E|F|G etc

meaning: a letter is defined as A or B or C or D or E or F or G etc.

Don't worry if you are unsure exactly how this works. The following standard structures are given in BNF to show their original form, but this relates closely to the equivalent BASIC. And in each case you will see detailed examples, too.

REPETITION TECHNIQUES

Pascal provides three techniques for repetition, and the choice is often dependent on the data that is present. All of these will be familiar to BBC BASIC programmers as the WHILE...DO, REPEAT...UNTIL and FOR...NEXT. The latter statement exists in other BASICs, too, and the first two can be simulated.

The article on structured programming (pages 173 to 178) shows how each of these works. The important difference between the three forms is the control condition that affects the number of repetitions. In the WHILE...DO, repetition continues while a known condition holds true. In the REPEAT...UNTIL it goes on until a condition occurs. These techniques are used when the number of repeats is not known beforehand. Where it is known then FOR...NEXT is often preferred.

Refer back to the flow diagrams on pages 173 to 178 if you need to see this diagrammatically—particularly if you are not familiar with BBC BASIC. Then look at the forms in Pascal.

WHILE...DO

In BNF, this takes the general form:

while <logical statement> **do** <statement>

This type of loop is usually used when in some circumstances the loop is not changed at all and the number of repetitions is not known beforehand. The repetition will continue once each time a known condition specified in the program remains true.

If you want to repeat a number of statements within the loop these statements are made into a single 'compound statement' by putting a begin and end at the beginning and end of a group of statements:

```
program example2(input,output);
{evaluate a running sum, using while}
var no,sum:integer;
begin
   sum := Ø;
   read(no);
   while no <> Ø do
   begin
```

```
      sum := sum + no;
      read(no)
   end; {while}
   writeln(sum)
end. {example2}
```

This example should be simple to follow, and you might like to try writing the algorithm on which it is based.

REPEAT...UNTIL

This takes the form:

repeat < statement > **until** < logical statement >

Statements between repeat and until are repeated until the end condition is true. All the statements between the repeat and until are executed at least once and the test is carried out at the end. It is not necessary to have compound statements bracketed between begin and end as the repeat/until acts in the same way.

The running sum example could be written to allow the different statement:

```
program example3(input,output);
{evaluate a running sum, using repeat}
var no,sum:integer;
begin
   sum := Ø;
   repeat
   read(no);
   sum := sum + no;
until no = Ø;
write(sum)
end. {example3}
```

This program is obviously superior to example2 as it uses the structure of repeat advantageously whereas the while structure needs an additional read(no). To work out a running sum, it is better if the program enters the loop so that the repeat statement is executed. In example2 the program arrives at a solution by working around a structure that is not really suitable. example3 uses the best Pascal structure.

FOR

In Pascal, FOR loops can only go in steps of +1 or −1. They take the form:

```
program example4(input,output);
{evaluate a running sum, using for}
var no,sum,amount,i : integer;
begin
   read(amount);
   sum := Ø;
   for i: = 1 to amount do
   begin
      read(no);
      sum := sum + no
   end; {for}
   writeln(sum
end. {example4}
```

DECISION MAKING

Again, like BASIC, Pascal has statements to use if a decision has to be made between several outcomes. the first one is the if-then-else—which also occurs in BASIC. This is usually made when selection is one or other of two events. If, however, the selection is to be made from a number of events then the case statement is usually more appropriate. This is not found in this form in BASIC.

IF...THEN...ELSE

If < logical statement > **then** < statement > **else** < statement2 >

This structure is described in *INPUT* on pages 173–178. It says that the first statement is obeyed if the controlling expression is true, or if the expression is false the second statement is followed. You can see from the BNF diagram that the else is optional so an alternative to if-then-else is if-then.

CASE

The case statement allows a program to select one statement from a whole list of possibilities.

The general form of the case statement is:

```
case expression of
   c1 : statement;
   c2 : statement;
         ''
         ''
         ''
   cn : statement
end;
```

The value of expression must correspond to one of the cs, and must be of the same type. In standard Pascal, if the value selected does not correspond to a case value then case is undefined and an error usually occurs. Some implementations, however, would ignore the case statement.

Notice how there is an end associated with case that does not have a matching begin. You will find that this is one of the few statements that has this feature (it *is* consistent if you think about it). It does, however, make things difficult when you are trying to debug a program by counting begin, ends; although it does reinforce the value of labelling ends (by writing end; {case}). The statements associated with each label (c) may be compound statements, but must have the usual begin/end. Look at this example of case:

```
program example5(input,output);
{input a number, output day of week—uses
   case}
var dayno : integer;
begin
   readln(dayno);
   if (dayno > Ø) and (dayno < 8)
   then

   case dayno of

      1 : writeln('Workday Monday');
      2 : writeln('Workday Tuesday');
      3 : writeln('Workday Wednesday');
      4 : writeln('Workday Thursday');
      5 : begin
          writeln('Workday Friday');
          writeln('Payday Friday')
          end;
      6,7 : writeln('no work Weekend')
end {case}
   else writeln('Input must be an integer
    between 1 & 7')
end. {example5}
```

With the S-Pascal program the labels 1, 2, 3, 4, 5, 6, and 7 must be put in brackets. The case statement will look like:

```
case dayno of
(1) :Writeln ('Workday Monday');
etc.
(6,7) : Writeln ('no work Weekend)
end {case}
```

This is non-standard Pascal but was necessary in this implementation because of the BASIC systems.

The program example5 illustrates how you can overcome the problem of attempting to enter a case statement when one of its values is undefined. Here we have used an if-then-else. The example also shows you how either of two values of dayno (in this example) may be used as a selector and that compound statements may be used for any selected case value.

DEVELOPING PROGRAMS

With a more detailed knowledge of the procedures available in Pascal, you can start to take a closer look at a Pascal program, remembering that you need to work this out in detail before you start work at the computer.

The example program deals with testing to see whether a string is a palindrome—a word such as radar, or a phrase such as 'madam i'm adam' which reads the same forward as it does backwards, ignoring all spaces and punctuation marks. Suppose you have a string that you want to test, which is:

```
the cat sat on the mat
```

The initial algorithm could read:

```
begin
   read in input string
   test if palindrome
end
```

This consists of just two steps, plus the beginning and end statement required by Pascal. In detail step 1 could read:

```
begin
   read in number of characters (n)
   read in input string into array A ,A . . . A
end
```

This could be refined to Pascal—to give a step (1):

```
readln(n);
for i: = 1 to n do read (a[i]);
```

A detailed algorithm for step 2 could read:

```
begin
   assign fwd to 1
   assign bwd to n
   while fwd < bwd do
      if a[fwd] is punctuation mark then
      increment fwd
      else
         if a[bwd] is punctuation mark then
         decrement bwd
      else
         if a[fwd] = a[bwd] then
            increment fwd
            decrement bwd
         else set boolean which (pal) to false
end
```

This could be refined to Pascal step 2 as:

```
      fwd := 1;
      bwd := n;
      while (fwd < bwd) and pal do
      if(a[fwd] = '')or(a[fwd] = '.')or(a[fwd] =
',')or(a[fwd] = '''')thenfwd: = fwd + 1
   else
   if(a[bwd] = '')or(a[bwd] = '.')or
(a[bwd] = ',')or(a[bwd] = '''') then bwd: =
bwd - 1
   else
   if a[fwd] = a[bwd] then
      begin
         fwd := fwd + 1;
            bwd := bwd -1
      end
      else pal := false
```

A detailed algorithm for step 3 could be:

```
if pal is true then write 'palindrome'
            else write 'not a palindrome'
```

which could be refined to Pascal step 3 as:

```
if pal then writeln (' palindrome');
      else writeln ('not a palindrome')
```

THE COMPLETE PROGRAM

Following on from this the complete program would read:

```
   program example 6(input,output);
   {the palindrome problem}
   const most = 3Ø;
   var a : array [1 . . most] or char;
      i,n,fwd,bwd : integer;
            pal : boolean;
         begin
            pal = true;
            readln(n);
            for i: = 1 to n do read (a[i]);
            fwd := 1;
            bwd := n;
            while (fwd < bwd) and pal do
               if(a[fwd] =  )or(a[fwd] = '.')or(a
[fwd] = ',')or(a[fwd] = '''') then fwd: = fwd + 1
            else
            if(a[bwd] = '')or(a[bwd] = '.')or(a
[bwd] = ',')or(a[bwd] = '''') then bwd: =
bwd - 1
            else
            if a[fwd] = a[bwd] then
            begin
               fwd := fwd + 1;
               bwd := bwd - 1;
            end {if}
            else pal = false;
      if pal then writeln ('palindrome')
            else writeln ('not a palindrome')
      end. {example6}
```

You'll notice that when we tested for a quote in the program it is expressed as four single quotes. Also, in the test for the Boolean variable (this just means a logical statement that isn't an arithmetic quantity and can be true or false), it is not necessary to say:

```
if pal = true then. . .
```

In other words:

```
if pal then. .
```

is sufficient.

IMPROVEMENTS

There are a number of improvements you can make to this program if your Pascal system has a few additional data types. The program—as written—should certainly work on most small integer-only Pascal systems.

The use of the Boolean flag (which is called pal in this example) is used as a method of leaving the while loop when it has been determined that a[fwd] does not equal a[bwd]. All that is done here is to set pal to false because the test a[fwd] = a[bwd] has failed (any Boolean variable of type *boolean* can only be true or false). When an attempt is made to go round the while loop again, the condition that requires pal to be true fails and the while loop is left. The technique used in BASIC might have been:

```
14Ø . . .
15Ø FOR I = 1 TO N
   . . .
   . . .
19Ø IF A(FWD) < > A(BWK) THEN 26Ø
   . . .
   . . .
23Ø NEXT I
24Ø PRINT "A PALINDROME"
25Ø GOTO 27Ø
26Ø PRINT "NOT A PALINDROME"
27Ø END
```

In BBC BASIC the code could have been a little more elegant but jumping out of the middle of the loop is certainly not structured.

It would have been a lot better if the algorithm had said:

```
begin
   define set of punctuation marks
   . . .
   . . .
   if a[fwd] in punctuation then increment fwd
   else
   if a[bwd] in punctuation then decrement
   backwards
   else
   . . .
```

A statement like this would have avoided the need to keep asking:

```
   if (a[fwd] = ' ') or (a[fwd] = ',') or
(a[fwd[ = '.') or (a[fwd] = '''') then . . .
```

Although you have not yet seen it being used, Pascal does in fact allow sets to be used—although some of the smallest compilers do not support the statement.

If your Pascal supports data type sets you could modify example 6 to look like:

```
program example7 (input,output);
{the palindrome problem — using sets}
const most = 3Ø;
var a : array[1 . . most] of char;
         i,n,fwd,bwd : integer;
         pal : boolean;
         punct : set of [' ','.',',',''''];
   begin
      pal := true;
      readln(n);
      for i: = 1 to n do read(a[i]);
      fwd := 1;
      bwd := n;
      while (fwd < bwd) and pal do
      if a[fwd] in punct then fwd: = fwd + 1
    else
      if a[bwd] in punct then
    bwd: = bwd – 1
    else
      if a[fwd] = a[bwd] then
      begin
      fwd := fwd + 1;
      bwd := bwd – 1;
    end
    else pal := false;
    if pal then writeln ('palindrome')
      else writeln ('not a palindrome')
end. {example7}
```

In the palindrome solution, the three outline steps are:

```
input string
ascertain if string is a palindrome
output result
```

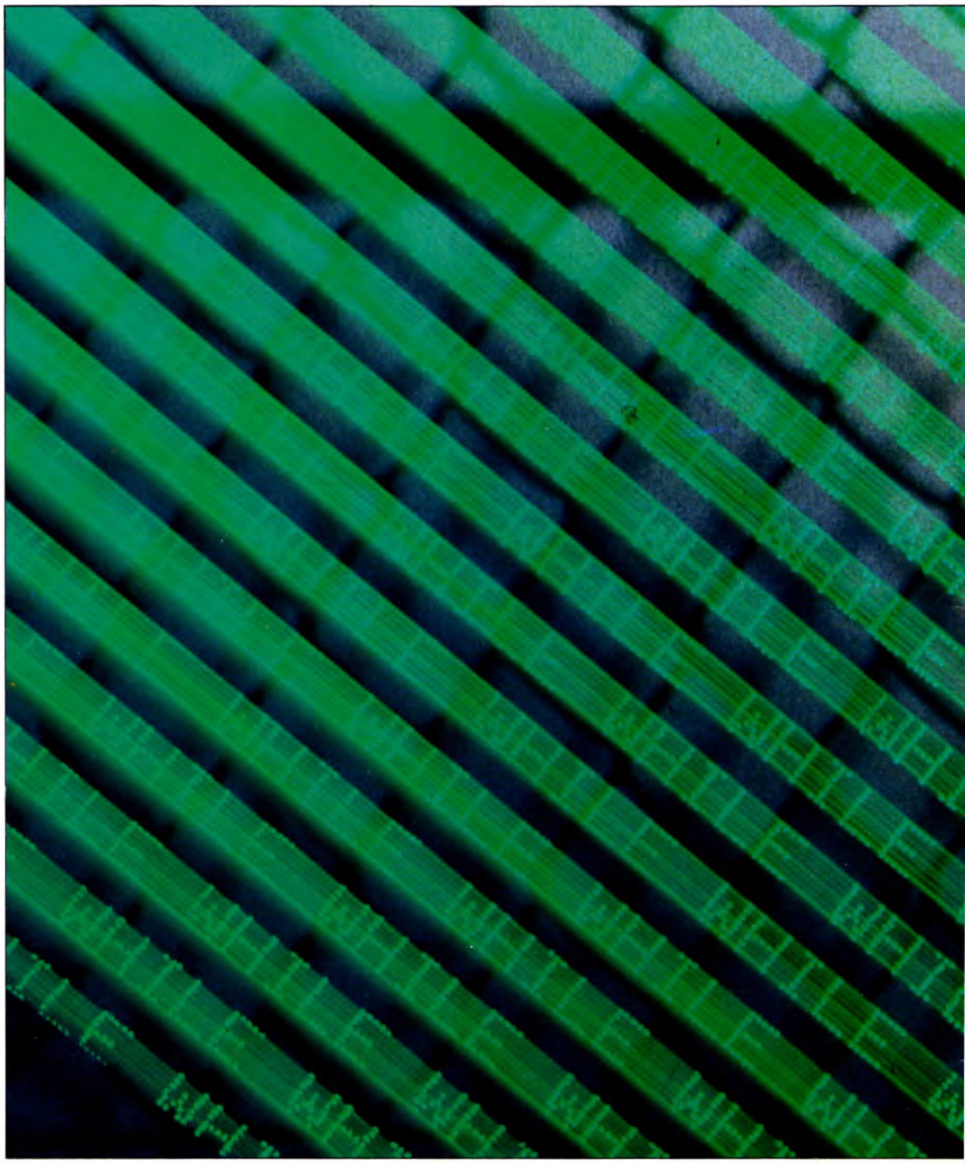
Each step was then refined into further steps until the result was Pascal code which was then pieced together to form a complete solution. For a small program this is fine, but in cases where the outline steps are large and complex it is best to test each of the independent modules initially and then call each module by a procedure call. You can then write a main program which simply consists of a number of procedure calls. A detailed description of how to do this is contained in *INPUT* pages 201 to 207 but bear in mind that with Pascal it is better to use the structured technique of stepwise refined algorithms rather than flowcharts.

PASCAL SYSTEMS

There are two main groups of Pascal systems. One group uses native code compilers—compilers written specifically for the machines that are to run the system. Obviously, many machines are very similar, using CP/M or MS DOS operating systems, so only very minor modifications are needed from one machine to another.

The other system is known as UCSD (University of California San Diego) Pascal. In this system the Pascal code is compiled down to the code of a hypothetical computer known as a p machine, and the completed code is known as p code. This means that the compiler that is necessary for the compilation is the same irrespective of the host machine. In fact, all of the UCSD system is, itself, written in Pascal, and each host machine has a simple program written specifically for it.

The advantage of UCSD Pascal is increased portability, but the disadvantage is that the generated code is slower in execution than a native code compiler.

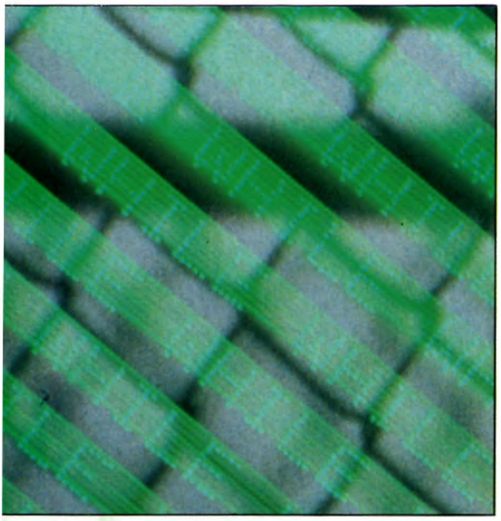


ENTERING PASCAL

If you have a Pascal system for your home computer you can try keying in one or more of the examples. When you have typed the source code, compile it and run the program. These programs should work for any Pascal system accepting lower case input. For the Electron and BBC using the S-Pascal pack the procedure is as follows (written as an algorithm):

```
If you have disk drives
then
begin
  place disk in drive;
  close hatch
  press shift;
  while holding shift do
  begin
    press and release break
    release shift
  end;{while}
else
  begin
    load cassette in recorder
    rewind cassette
    type CHAIN "S.PASCAL"
  end;{else}
  type*NEW return
  key in your program
  press escape
end.{s-pascal test}
```

You will notice that using this system, line numbers will appear. This is to allow the Acorn editor to be used. Pascal itself does not use line numbers, but most of the home computer versions offer editing facilities to



make them simpler to use.

To edit your newly created program type:

```
*edit
```

To load a file type:

```
LOAD "filename" after *edit
```

To save your file type:

```
SAVE "filename"after *edit
```

Having typed in your program, checked it and saved it, try compiling it with the instruction:

```
*compile.
```

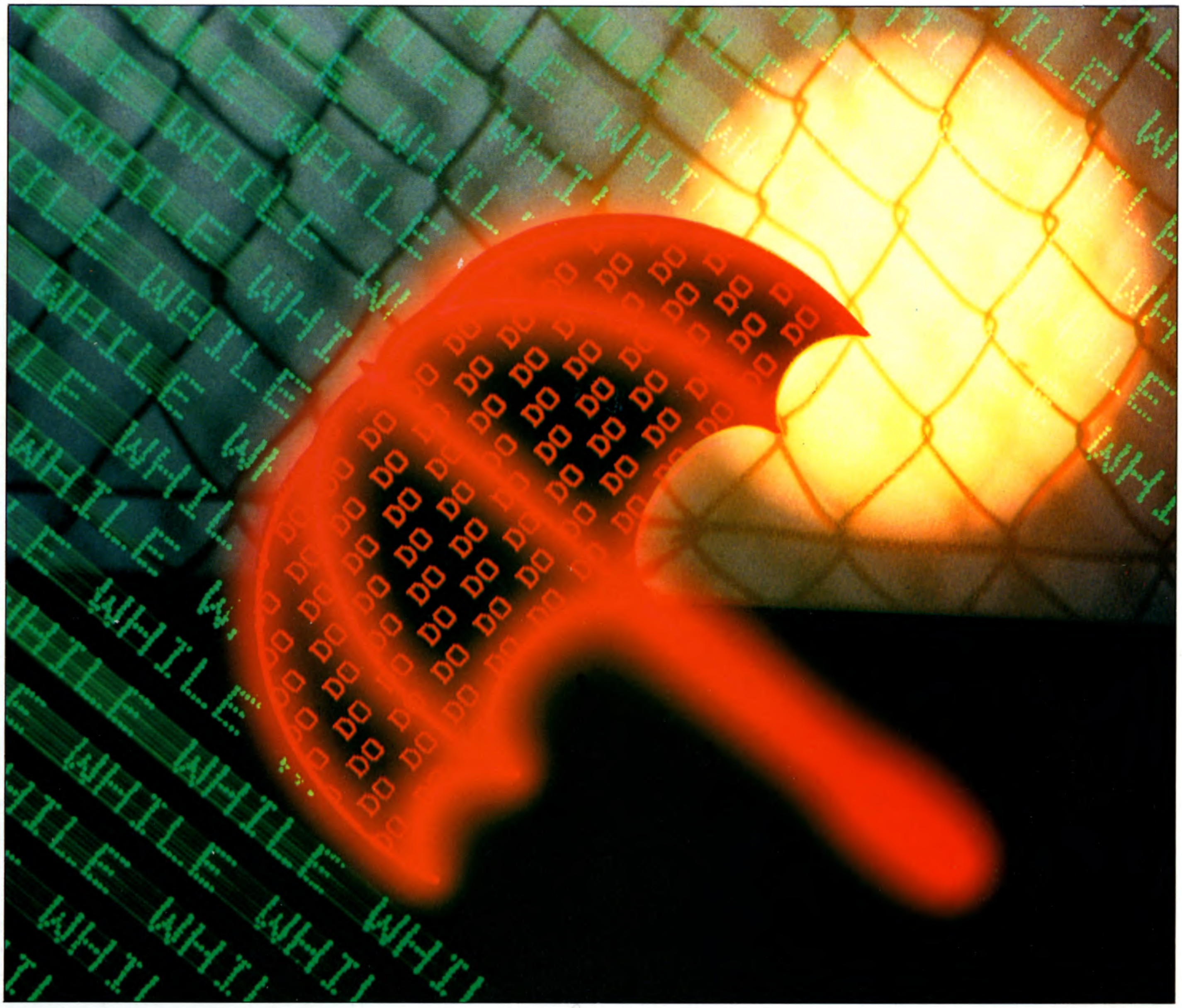
If there is an error the compilation will stop and the approximate position of the error will appear. You will then need to edit out the error using the normal editor. Don't forget *edit first.

If you have managed to compile the source code so that no error messages appear then run your program with:

```
*go
```

Perhaps the two most common causes of errors are either variables omitted in the var declaration, or missing or wrongly placed semi-colons.

Remember each statement must have a semi-colon after it except before an end. A compound statement is surrounded by begin/end (no semi-colon after begin). If you put a semi-colon immediately after a while, repeat or for statement you are marking that as the end of the statement.



FROM BASIC TO BEETHOVEN

Continue entering the program that will minimise your composing effort. If music is your forté, stave off creativity 'till part three when the program is completed and explained

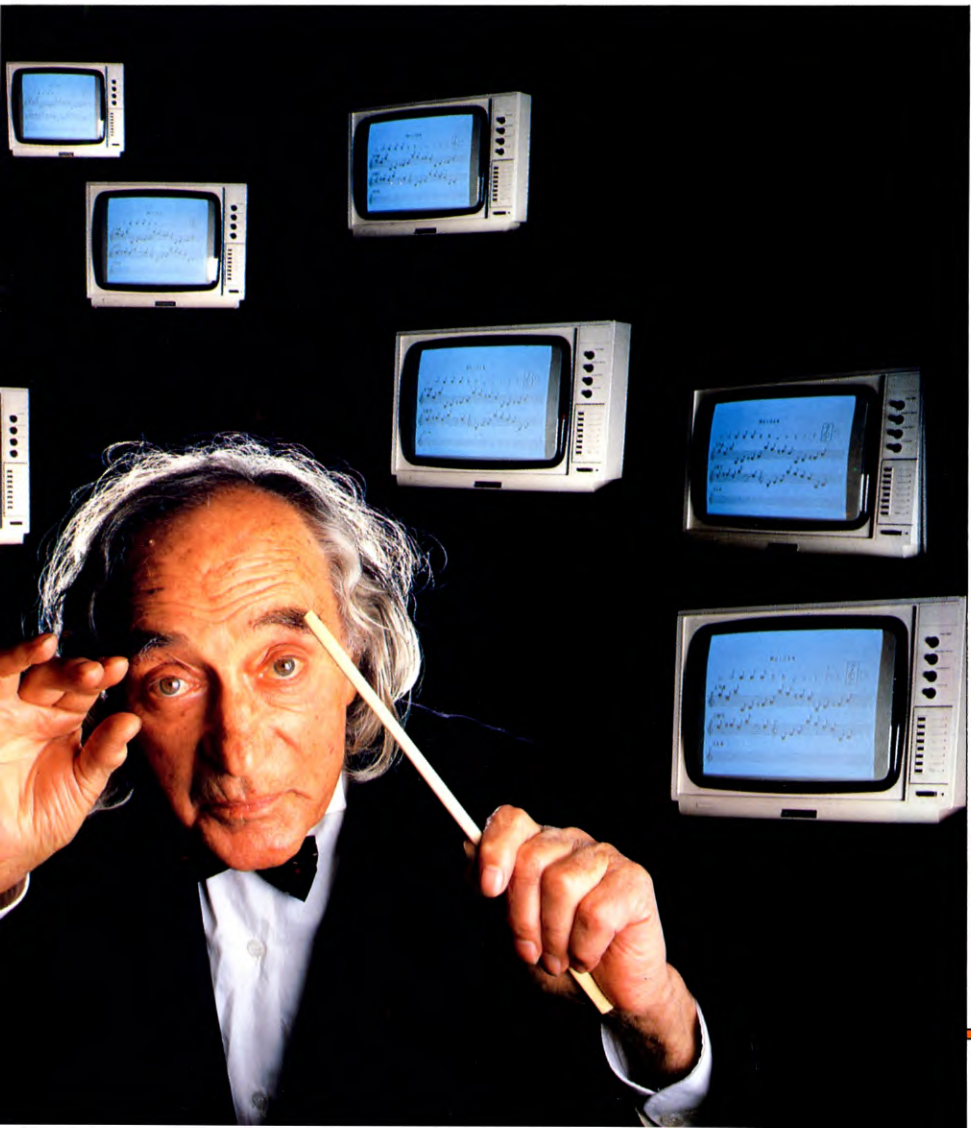
Here is the second part of the music composer program. LOAD in part one and add the section given here. Then SAVE it again ready for the final part next time. The next part will also give full instructions on how to use each of the programs.

As you type in the programs you'll see that they consist of several sections or routines called from the main menu. The Acorn is the exception in that it goes straight into the graphics display and the routines are initiated by different keypresses.

You cannot RUN the whole program until you type in the last part but you can get a good idea of what each program does by having a look at the menu or the Acorn's graphics. So even though you haven't entered the whole program, typing RUN should still give you the main display to see what it looks like. (On the Commodore, deleting Line 14Ø will allow the menu to appear, but don't forget to reinstate it afterwards before you SAVE the program.)

```
2ØØØ CLS
2Ø1Ø GOSUB 25ØØ
214Ø PRINT 'ct;"□Notes entered
   (";maxnotes; "Max.)"'
216Ø INPUT "Enter Notes — <RET> to
   end ";N$
2175 IF LEN (N$)=Ø THEN GOTO 19Ø
218Ø FOR i=1 TO LEN (N$): IF (N$(i)<"Ø"
   OR N$(i)>"9") AND (N$(i)<>"—")
   THEN GOTO 2ØØØ
219Ø LET N=VAL (N$)
22ØØ IF INT (N/1ØØØ)>11 AND INT
   (N/1ØØØ)<>—4 THEN GOTO 2ØØØ
221Ø IF N<Ø THEN GOTO 228Ø
222Ø LET M=INT (N/1ØØ): LET
   D=N—M*1ØØ
223Ø LET O=M—INT (M/1Ø)*1Ø: IF O<1
   OR O>7 THEN GOTO 2ØØØ
224Ø LET M=INT (M/1Ø)+(O—1)*
   12—36
226Ø LET ct=ct+1: LET t(2*ct—1)=D: LET
   t(2*ct)=M
227Ø GOTO 2ØØØ
228Ø LET M=—4: LET D=Ø—(N—M*1ØØ)
229Ø IF M<>—4 THEN GOTO 2ØØØ
231Ø GOTO 226Ø
25ØØ PRINT "Enter Note as <Number>,
   <Octave>,<Duration>."'
252Ø PRINT "C — Ø E — 4 G# —
   8"'"C# — 1 F — 5 A — 9"'"D — 2
   F# — 6 A# — 1Ø"'"D# — 3 G — 7
   B — 11"
256Ø PRINT '"REST is —4"'"Semi-
   Quaver=1 Minim□□□□□□=8"'
   "Quaver□□□□□□=2 Semi-
   Breve□=16"'"Crotchet□□□=4
   Octave□=1 to 7"
26ØØ PRINT '"Duration MUST be 2
   digits"'"e.g. 33Ø4 — D#,3rd Oct.
   Crotchet"'"e.g. 64Ø8 — F#,4th Oct.
   Minim"
263Ø RETURN
3ØØØ CLS
3Ø1Ø PRINT "Tune Replay"
3Ø2Ø PRINT : PRINT : PRINT
3Ø3Ø PRINT "Enter Tempo□—□□
   (1—15)□□";
3Ø4Ø INPUT S
3Ø5Ø IF S<1 OR S>15 THEN GOTO 3ØØØ
3Ø6Ø LET tempo=Ø.Ø2*(16—S)
3Ø7Ø FOR i=1 TO ct
3Ø8Ø LET D=t(2*i—1): LET M=t(2*i)
3Ø9Ø IF m=—4 THEN GOTO 312Ø
31ØØ BEEP D*tempo,M
311Ø GOTO 313Ø
312Ø PAUSE 5Ø*D*tempo
313Ø NEXT i
314Ø RETURN
4ØØØ CLS
4Ø1Ø PRINT "Tune Editor"''"D — Display all
   notes"'"E — Edit a note"'"I — Insert a
   Note"'"X — Delete a note"''"R —
   Return to Main Menu"
4Ø9Ø PRINT "Enter Option — ";
41ØØ LET O$=INKEY$: IF O$="" THEN
   GOTO 41ØØ
41Ø5 IF CODE (O$)<97 THEN LET
   O$=CHR$ (CODE (O$)+32)
411Ø IF O$<>"d" AND O$<>"e" AND
   O$<>"i" AND O$<>"r" AND
   O$<>"x" THEN GOTO 4ØØØ
412Ø IF O$="r" THEN RETURN
413Ø IF O$="e" THEN GOTO 43ØØ
4134 IF O$="i" THEN GOTO 47ØØ
4136 IF O$="x" THEN GOTO 48ØØ
414Ø CLS
415Ø FOR i=1 TO ct
416Ø LET M=t(2*i): LET D=t(2*i—1)
417Ø LET O=INT ((M+36)/12)+1
418Ø LET N=(M+36)—(O—1)*12
419Ø PRINT i;"□Note—□";N;"□Oct.—
   □";O;"□Dur.—□";D
4195 POKE 23692,255
42ØØ IF i=2Ø*INT (i/2Ø) THEN GOTO 422Ø
421Ø GOTO 425Ø
422Ø PRINT '"Any Key to Continue□";
423Ø PAUSE Ø
424Ø PRINT
425Ø NEXT i
426Ø PRINT '"Any Key to Return□";
427Ø PAUSE Ø
428Ø GOTO 4ØØØ
432Ø CLS
433Ø GOSUB 25ØØ
```

```
2908 FOR V=Ø TO 2
291Ø N=VOICE(V,CT)
2911 IF N=−1 THEN 2926
2912 IF INT(N/1Ø)+1=−2 THEN 2927
2913 POKE SID+(V*7)+4,CR(V)
2914 O=N−(INT(N/1Ø)*1Ø)
2915 IF O>7 THEN O=7
2916 M=12*O+INT(N/1Ø)
2917 IF M=Ø THEN 2928
2918 HF =H(M):LF=L(M)
292Ø POKE SID+(V*7),LF:POKE
    SID+(V*7)+1,HF
2922 POKE SID+(V*7)+4,CR(V)+1
2924 GOTO 2928
2926 POKE SID+(V*7)+4,CR(V):
    FOR I=1 TO 5Ø:NEXT:GOTO 2928
2927 DE=VAL(RIGHT$(STR$(N),1)):
    FOR I=1 TO(2↑(DE−1))*5Ø:
    NEXT I
2928 NEXT V
293Ø FOR I=1 TO TEMPO:NEXT I
2932 CT=CT+1:IF CT>5ØØ THEN RETURN
2934 GOTO 29Ø4
3ØØØ PRINT "♥VOICE□";A;
    "□PARAMETERS"
3ØØ5 R=Ø
3Ø2Ø PRINT"▣▣WAVEFORM"TAB(17)
    "(T,S,P,N)"TAB(28);:Z$="TSPN"
3Ø3Ø PRINT MID$(Z$,WF(A−1)+1,1)
3Ø8Ø PRINT"ATTACK"TAB(17)"(Ø−15)"TAB
    (28);AT(A−1)
3Ø9Ø PRINT"DECAY"TAB(17)"(Ø−15)"TAB
    (28);DE(A−1)
31ØØ PRINT"SUSTAIN"TAB(17)"(Ø−15)"
    TAB(28);SU(A−1)
311Ø PRINT"RELEASE"TAB(17)"(Ø−15)"
    TAB(28);RE(A−1)
312Ø PRINT"FREQUENCY"TAB(17)
    "(Ø−65535)"TAB(28);FR(A−1)
313Ø PRINT"PULSE WIDTH"TAB(17)
    "(Ø−4Ø95)"TAB(28);PW(A−1)
314Ø PRINT"SYNCHRONISATION"TAB(17)
    "(ON/OFF)"TAB(28);
315Ø IF SY(A−1)=1 THEN 316Ø
3155 PRINT"OFF":GOTO 317Ø
316Ø PRINT"ON"
317Ø PRINT"RING MODULATION"TAB(17)
    "(ON/OFF)"TAB(28);
318Ø IF RM(A−1)=1 THEN 319Ø
3185 PRINT"OFF":GOTO 32ØØ
319Ø PRINT"ON"
32ØØ PRINT"FILTER"TAB(17)"(ON/OFF)"TAB
    (28);
321Ø IF FI(A−1)=1 THEN 322Ø
3215 PRINT"OFF":GOTO 3222
322Ø PRINT"ON"
3222 PRINT"▣RETURN TO MAIN MENU"
3226 PRINT"▣▣▣USE CURSOR
    UP/DOWN TO MOVE ARROW"
3228 PRINT"<RETURN> TO SELECT"
323Ø PRINT "▤"TAB(36);
3235 FOR I=1 TO 14:PRINT
    "▣□▮▮";:NEXT
3236 PRINT "▤"TAB(36);
3237 FOR I=1 TO LINE:PRINT
    "▣□▮▮";:NEXT
3238 PRINT "←";
324Ø GET B$:IF B$="" THEN 324Ø
325Ø IF B$=CHR$(17) THEN
    LINE=LINE+1
326Ø IF B$=CHR$(145) THEN
    LINE=LINE−1
327Ø IF LINE<3 THEN LINE=3
328Ø IF LINE=13 AND B$=CHR$(17) THEN
    LINE=14
3282 IF LINE=13 AND B$=CHR$(145)
    THEN LINE=12
3285 IF LINE>14 THEN LINE=14
329Ø IF B$=CHR$(13) THEN GOSUB 5ØØØ
33ØØ IF R=1 THEN RETURN
33Ø5 IF R=2 THEN 3ØØØ
```



```
3308 GOTO 3230
4000 PRINT “♡GENERAL PARAMETERS”
4005 R=0
4020 PRINT“▣▣FILTER FREQUENCY”TAB
   (20)“(0–2047)”TAB(29);FF
4030 PRINT“FILTER RESONANCE”TAB(20)
   “(0–15)”TAB(29);FR
4040 PRINT“FILTER LOW PASS”TAB(20)
   “(ON/OFF)”TAB(29);
4050 IF LP=1 THEN 4060
4055 PRINT“OFF”:GOTO 4070
4060 PRINT“ON”
4070 PRINT“FILTER BAND PASS”
   TAB(20)“(ON/OFF)”TAB(29);
4080 IF BP=1 THEN 4090
4085 PRINT“OFF”:GOTO 4100
4090 PRINT“ON”
4100 PRINT“FILTER HIGH PASS”
   TAB(20)“(ON/OFF)”TAB(29);
4110 IF HP=1 THEN 4120
4115 PRINT“OFF”:GOTO 4130
4120 PRINT“ON”
4130 PRINT“VOICE 3 CONNECTION”
   TAB(20)“(ON/OFF)”TAB(29);
4140 IF V3=1 THEN 4150
4145 PRINT“ON”:GOTO 4155
4150 PRINT“OFF”
4155 PRINT“VOLUME”TAB(20)“(0–15)”
   TAB(29)VOL
4158 PRINT“TEMPO”TAB(20)“(0–20)”TAB
   (29)(200–TEMPO)/10
4160 PRINT“▣RETURN TO MAIN MENU”
4170 PRINT“▣▣▣▣▣USE CURSOR
   UP/DOWN TO MOVE ARROW”
4175 PRINT“<RETURN> TO SELECT”
4180 PRINT “▤”TAB(36);
4190 FOR I=1 TO 12:PRINT
   “▣□▮▮”;:NEXT I
4200 PRINT “▤”TAB(36);
4210 FOR I=1 TO LINE:PRINT
   “▣□▮▮”;:NEXT I
4220 PRINT“←”;
4230 GET B$:IF B$=“” THEN 4230
4240 IF B$=“▣” THEN LINE=LINE+1
4250 IF B$=“◘” THEN LINE=LINE–1
4260 IF LINE<3 THEN LINE=3
4270 IF LINE=11 AND B$=“▣” THEN
   LINE=12
4280 IF LINE=11 AND B$=“◘” THEN
   LINE=10
4290 IF LINE>12 THEN LINE=12
4300 IF B$=CHR$(13) THEN GOSUB 5000
4310 IF R=1 THEN RETURN
4320 IF R=2 THEN 4000
4330 GOTO 4180
5000 IF A=4 THEN 5500
5010 IF LINE<>14 THEN 5025
5020 R=1:RETURN
5025 R=2
5030 ON LINE–2 GOTO 5040,5130,5130,
   5130,5130,5220,5270,5320,5360,5400
5040 GOSUB 5900
5050 PRINT“ENTER T,S,P OR N –□□”;
5060 GET C$:IF C$=“” THEN 5060
5070 IF C$<>“T” AND C$<>“S” AND
   C$<>“P” AND C$<>“N” THEN
   RETURN
5080 IF C$=“T” THEN WF(A–1)=0
5090 IF C$=“S” THEN WF(A–1)=1
5100 IF C$=“P” THEN WF(A–1)=2
5110 IF C$=“N” THEN WF(A–1)=3
5120 GOSUB 6000:RETURN
5130 GOSUB 5900
5140 PRINT“ENTER VALUE –□□”;
5150 INPUT V
5160 IF V<0 OR V>15 THEN RETURN
5170 IF LINE=4 THEN AT(A–1)=V
5180 IF LINE=5 THEN DE(A–1)=V
5190 IF LINE=6 THEN SU(A–1)=V
5200 IF LINE=7 THEN RE(A–1)=V
5210 GOSUB 6000:RETURN
5220 GOSUB 5900
5230 PRINT“ENTER VALUE –□□”;
5240 INPUT V
5250 IF V<0 OR V>65535 THEN RETURN
5260 FR(A–1)=V:GOSUB 6000:RETURN
5270 GOSUB 5900
5280 PRINT“ENTER VALUE –□□”;
5290 INPUT V
5300 IF V<0 OR V>4095 THEN RETURN
5310 PW(A–1)=V:GOSUB 6000:RETURN
5320 IF SY(A–1) THEN 5340
5330 SY(A–1)=1:GOTO 5350
5340 SY(A–1)=0
5350 GOSUB 6000:RETURN
5360 IF RM(A–1) THEN 5380
5370 RM(A–1)=1:GOTO 5390
5380 RM(A–1)=0
5390 GOSUB 6000:RETURN
5400 IF FI(A–1) THEN 5420
5410 FI(A–1)=1:GOTO5430
5420 FI(A–1)=0
5430 GOSUB 6000:RETURN
5500 IF LINE<>12 THEN 5515
5510 R=1:RETURN
5515 R=2
5520 ON LINE–2 GOTO 5530,5580,5650,
   5690,5730,5770,5580,5810
5530 GOSUB 5900
5540 PRINT“ENTER VALUE –□□”;
5550 INPUT V
5560 IF V<0 OR V>2047 THEN RETURN
5570 FF=V:GOSUB 6000:RETURN
5580 GOSUB 5900
5590 PRINT“ENTER VALUE –□□”;
5600 INPUT V
5610 IF V<0 OR V>15 THEN RETURN
5620 IF LINE=4 THEN FR=V
5630 IF LINE=9 THEN VOL=V
5640 GOSUB 6000:RETURN
5650 IF LP THEN 5670
5660 LP=1:GOTO5680
```

```
500 DEFPROCSave
510 VDU4:CLS:PRINTTAB(6,3);“Enter the
   output file name. ”TAB(14,4);:INPUTFn$
520 IFFn$=“”GOTO570
530 X=OPENOUTFn$
540 PRINT#X,NO%,SD,EV%
550 FORY=0TONO%–1:PRINT#X,S(0,Y),
   S(1,Y)
560 NEXT:CLOSE#X
570 PROCno:ENDPROC
580 DEFPROCLoad
590 VDU4:CLS:PRINTTAB(6,3);“Enter the
   input file name. ”TAB(14,4);:INPUTFn$
600 IFFn$=“”GOTO 570
610 X=OPENINFn$:INPUT#X,NO%,SD,EV%
620 FORY=0TONO%–1:INPUT#X,S(0,Y),
   S(1,Y)
630 NEXT:CLOSE#X
640 D%=NO%–29:IFD%<1THEND%=1
650 PROCDisplay(D%,NO%):ENDPROC
660 DEFPROCEnvelope
670 VDU4:CLS:PRINTTAB(4,2)“1:Organ.”TAB
   (19,2)“4:Harpsichord.”
680 PRINTTAB(4,3)“2:Vibrato(1).”TAB(19,3)
   “5:Piano.”
690 PRINTTAB(4,4)“3:Vibrato(2).”
700 PRINTTAB(13,5)“Enter choice.”
710 REPEAT:VDU7:EV%=GET–48:UNTIL
   EV%>0ANDEV%<6
720 PROCno:ENDPROC
730 DEFPROCV
740 VDU4:CLS:PRINTTAB(7,2)“Enter the note
   number that”TAB(8)“you want to
   display from?”;:PROCRead
750 IFD%>=NO%PROCrange(1):GOTO740
760 IFD%=0D%=1
770 PROCDisplay(D%,NO%):ENDPROC
780 DEFPROCChangespeed
790 REPEAT:SOUND0,4,6,4*SD
800 TIME=0:REPEATUNTILTIME>SD*20
810 A%=INKEY(1):*FX15,0
820 IFINKEY(–58)SD=SD–0.125:
   IFSD<1SD=1
830 IFINKEY(–42)SD=SD+0.125:IF
   SD>3SD=3
840 UNTILA%=13:SOUND16,0,0,0:A%=67:
   ENDPROC
850 DEFPROCTie(Q%,W%)
860 D%=S(1,Q%)
870 IFD%<>11ANDD%<>7ANDD%<>
   4ANDD%<>2ANDD%<>1ENDPROC
880 VDU4:CLS:PRINTTAB(12,3);“Tieing”
890 IFI%>4I%=0
900 O%=D%(I%):PROCM(0,4)
910 IFA%>13ANDA%<91GOTO990
920 IFD%(I%)>D%ANDD%=4D%=3
930 IFD%(I%)>D%ANDD%=7D%=4
940 IFD%(I%)>D%S(1,Q%)=D%+D%(I%)
   ELSES(1,Q%)=D%+5–I%
```

```
950 VDU5:PROCDrawnote(D%(I%),SE% + NS
   (NE))
960 Y% = SE% + NS(NE)
970 MOVEX% − S%*2 + 15,Y% − 32
980 FORX = ØTO2Ø:DRAWX%(X) + X% − 73,
   Y%(X) + Y% − 32:NEXT
990 PROCno:A% = 84:ENDPROC
1000 DEFPROCInsert
1010 VDU4:CLS:PRINTTAB(7,2)“Enter the note
   number that”TAB(8)
   “you want to insert after?”;:
   PROCRead
1020 IFD$ = “”THENPROCno:ENDPROC
1030 IFD% > = NO%PROCrange(1):
   GOTO1010
1040 TF% = F%:F3% = 1
1050 K% = NO% − 1 − D%:NO% = D% + 1:
   LT% = LT% − K%
1060 FORX = K% − 1TOØSTEP − 1
1070 S(Ø,LT% + 1 + X) = S(Ø,NO% + X):S(1,
   LT% + 1 + X) = S(1,NO% + X)
1080 NEXT:J% = D% + 1:PROCDisplay
   (NO% − 2Ø,NO%)
1090 REPEAT:J% = J% − 1
1100 UNTILS(Ø,J%) = 256ORS(Ø,J%) = 257
1110 F% = (S(Ø,J%) < > 256)
1120 ENDPROC
1130 DEFPROCInsertexit
1140 VDU4:CLS:PRINTTAB(7,2)“Exiting from
   insert mode”
1150 IFNO% = LT% + 1THEN118Ø
1160 FORX = ØTOK% − 1
1170 S(Ø,NO% + X) = S(Ø,LT% + 1 + X):S(1,
   NO% + X) = S(1,LT% + 1 + X):NEXT
1180 NO% = NO% + K%:LT% = TLT%:F% =
   TF%:F3% = Ø
1190 PROCDisplay(NO% − 2Ø,NO%)
1200 ENDPROC
1210 DEFPROCDelete
1220 VDU4:PRINTTAB(15,6)“No:”;NO%
   TAB(7,2)“Enter the note number
   that”TAB(8)“you want to delete from?□”;
1230 PROCRead
1240 IFD$ = “”PROCno:ENDPROC
1250 L% = D%
1260 IFL% < > ØGOTO129Ø
1270 IFS(Ø,Ø) = 256S(Ø,Ø) = 257ELSE
   S(Ø,Ø) = 256
1280 PROCDisplay(E%(Ø),NO%):ENDPROC
1290 IFL% > = NO%PROCrange(1):GOTO
   122Ø
1300 CLS:PRINTTAB(7,2)“Enter the note
   number that”TAB(7)“you want to delete
   to?□”;:PROCRead
1310 IFD$ = “”R% = L%:GOTO135Ø
1320 IFD$ = “999”R% = NO% − 1:
   GOTO135Ø
1330 R% = D%:IFR% > = NO%PROCrange(1):
   GOTO13ØØ
1340 IFR% < L%PROCrange(Ø):GOTO13ØØ
1350 M% = R% − L% + 1:IFR% = NO% − 1
   THEN139Ø
1360 D% = L% − 1:REPEAT:D% = D% + 1
1370 S(Ø,D%) = S(Ø,M% + D%):S(1,
   D%) = S(1,M% + D%)
1380 UNTILD% + M% = NO%
1390 NO% = NO% − M%:IFNO% > = E%(Ø)
   D% = E%(Ø)ELSED% = NO% − 29:IF
   D% < 1D% = 1
1400 PROCDisplay(D%,NO%)
1410 C% = NO%:REPEAT:C% = C% − 1:UNTIL
   S(Ø,C%) = 256ORS(Ø,C%) = 257
1420 IFS(Ø,C%) = 256F% = ØELSEF% = −1
1430 ENDPROC
1440 DEFPROCDrawtrebleclef(Y%)
1450 FORD% = Y% + 96TOY%STEP − 32:
   MOVEX%,D%
1460 PRINTT$((Y% + 96 − D%)/32):NEXT
1470 X% = X% + S% + 2Ø:ENDPROC
1480 DEFPROCDrawbassclef(Y%)
1490 FORD% = Y% + 6ØTOY% + 28STEP −
   32:MOVEX%,D%
1500 PRINTB$((Y% + 6Ø − D%)/32):NEXT
1510 X% = X% + S% + 2Ø:ENDPROC
1520 DEFPROCDrawstave(D%)
1530 FORY% = D% + 64TOD%STEP − 16
1540 MOVE2Ø,Y%:DRAW125Ø,Y%
1550 NEXT:ENDPROC
1560 DEFPROCDrawrest(R%,Y%)
1570 MOVEX%,Y%:PRINTR$(R%,Ø)
1580 MOVEX%,Y% − 32:PRINTR$(R%,1)
1590 X% = X% + S%:ENDPROC
1600 DEFPROCDrawnote(N%,Y%)
1610 MOVE X%,Y%
1620 IFN% > 6PRINTN1$ELSEPRINTN2$
1630 IFN% < 11MOVEX% + 28,Y% − 16:DRAW
   X% + 28,Y% + 32
1640 IFN% < 4DRAWX% + 42,Y% + 2Ø
1650 IFN% = 1MOVEX% + 3Ø,Y% + 2Ø:DRAW
```

```
   X%+42,Y%+8
1660 IFNE<3MOVEX%-7,SE%-16:
   DRAWX%+34,SE%-16
1670 IFNE=ØMOVEX%-7,SE%-32:DRAW
   X%+34,SE%-32
1680 IFNE>13MOVEX%-7,SE%+8Ø:DRAW
   X%+34,SE%+8Ø
1690 IFNE>15MOVEX%-7,SE%+96:DRAW
   X%+34,SE%+96
1700 X%=X%+S%
1710 IFN%=1ORN%=2ORN%=4
   ORN%=7ORN%=11ENDPROC
1720 D%=-1
1730 REPEAT:D%=D%+1:UNTIL
   N%-D%(D%)>Ø
1740 O%=N%-D%(D%)
1750 IFO%=5O%=11
1760 IFO%=4O%=7
1770 IFO%=3O%=4
1780 PROCDrawnote(O%,Y%)
1790 MOVEX%-S%*2+15,Y%-32
1800 FORX=ØTO2Ø:DRAWX%(X)+X%-73,
   Y%(X)+Y%-32:NEXT:ENDPROC
1810 DEFPROCACC(Acc$,Y%)
1820 MOVEX%,Y%:PRINTAcc$:X%=X%+
   S%-2Ø
1830 ENDPROC
1840 DEFPROCDisplaysymbols
1850 X%=1ØØ: S%=S%+2Ø: FOR X=Ø
   TO 4
1860 PROCDrawnote(D%(X),71Ø):NEXT
1870 PROCACC(S$,71Ø):X%=X%+2Ø
1880 PROCACC(F$,71Ø):X%=X%+2Ø
1890 FORX=ØTO4:PROCDrawrest(X,71Ø):
   NEXT
1900 S%=S%-2Ø:PROCDrawtrebleclef(67Ø)
1910 PROCDrawbassclef(67Ø):ENDPROC
1920 DEFPROCDrawthreestaves
1930 FORX=ST%(Ø)TOST%(2)STEP-2ØØ
1940 PROCDrawstave(X):NEXT:ENDPROC
```

```
600 P=P-36
610 IF P=1 ANDC<3 THEN C=C+1:
   OC$=MID$(STR$(C),2)
620 IF P=2 ANDC>1 THEN C=C-1:
   OC$=MID$(STR$(C),2)
630 IF (P=3 OR P=6)ANDLE>1 THENLE=
   LE-1:LE$=MID$(R1$,LE,1)+"□"
640 IF (P=4 OR P=7) ANDLE<5 THEN
   LE=LE+1:LE$=MID$(R1$,LE,1)+
   "□"
650 IF P=5 THENI$="p":P$="":
   GOTO7ØØ
660 IF P=6 OR P=7 THEN LE$=LEFT$
   (LE$,1)+"."
670 IF P=8 THEN RETURN
680 IF P=9 AND NN>Ø THEN NN=NN-1
690 GOTO49Ø
700 IF I$>="a" AND I$<="g" THEN
   P$=CHR$(ASC(I$)-32) ELSEIF
   I$<>"p" THEN P$=I$+"#"ELSE
   P$=""
710 PL$="V31;T"+STR$(TE)+"L"+L2$
   (INSTR(R1$,LEFT$(LE$,1)))
720 IF MID$(LE$,2,1)="." THEN
   PL$=PL$+"."
730 PL$=PL$+"O"+OC$+P$:PLAY PL$
740 NN=NN+1:N$(NN)=I$+OC$+LE$
750 GOTO 53Ø
760 CLS
770 PRINT@5,"MANUAL ENTRY OF
   NOTES":PRINT@34,"ENTER 'm' TO
   RETURN TO MENU"
780 PRINT"USE FORMAT: a1h. ('.'
   OPTIONAL)"
790 GOSUB 28Ø:PRINT@416,"ENTER NOTE
   STRING:"
800 IF NN=MX THEN PRINT@448,
   "MAXIMUM NOTES ENTERED!":FORD=1
   TO1ØØØ:NEXT:RETURN
810 POKE 329,Ø:PRINT@448:PRINT@462,;:
   LINE INPUT A$
820 IF A$="" THEN 81Ø
830 IF A$="m" OR A$="M" THEN
   RETURN
840 IF LEN(A$)>4 OR LEN(A$)<3 THEN
   91Ø
850 IF (INSTR(R3$,LEFT$(A$,1)))=Ø THEN
   91Ø
860 IF LEFT$(A$,1)="p" THEN A$="p"+
   "□"+MID$(A$,3):GOTO88Ø
870 IF (INSTR(R2$,MID$(A$,2,1)))=Ø
   THEN 91Ø
880 IF (INSTR(R1$,MID$(A$,3,1)))=Ø
   THEN 91Ø
890 IF MID$(A$,4,1)<>"." THEN A$=
   LEFT$(A$,3)+"□"
900 GOTO 92Ø
910 PRINT@448,LEFT$(A$,4);"□-
   ILLEGAL ENTRY!":SOUND 1,5:GOTO 81Ø
920 NN=NN+1:N$(NN)=A$
930 GOTO 79Ø
940 IF NN=Ø THEN RETURN ELSE POKE
   329,Ø
950 PRINT@448,"1: DELETE NOTES","2:
   INSERT NOTES","3: CHANGE NOTES","4:
   CONTINUE";
960 A$=INKEY$:IF A$<"1" OR A$>"4"
   THEN 96Ø
970 OP=VAL(A$)
980 ON OP GOTO 1ØØØ,11ØØ,126Ø,99Ø
990 RETURN
1000 IF NN=Ø THEN 94Ø ELSE CLS:INPUT
   "START AT WHICH NOTE";ST
1010 IF ST=Ø THEN 94Ø
1020 IF ST>NN THEN 1ØØØ
1030 PRINT@64,"DELETE HOW MANY
   (ENTER=1)";:INPUT ND
1040 IF ND<=Ø THEN ND=1
1050 IF ST+ND-1>NN THEN
   ND=NN-ST+1
1060 FOR I=1 TO ND
1070 FOR J=ST TO NN-1:N$(J)=
   N$(J+1):NEXT
1080 NN=NN-1
1090 NEXT I:RETURN
1100 CLS:PRINT@7,"INSERT NOTES MODE"
1110 PRINT@64,"START INSERT AFTER
   WHICH NOTE":INPUT ST
1120 IF ST<Ø THEN 94Ø
1130 IF ST>NN THEN ST=NN
1140 PRINT@64,"ENTER 'm' WHEN YOU'VE
   FINISHED":PRINT
1150 IF NN=MX THEN PRINT"MAX
   NUMBER OF NOTES ENTERED!":
   FORD=1TO1ØØØ:NEXT:RETURN
1160 INPUT N$:IF N$="M" OR N$="m"
   THEN RETURN
1170 IF LEN(N$)<3 THEN PRINT
   "ILLEGAL":GOTO116Ø
1180 IF INSTR(R3$,LEFT$(N$,1))=Ø OR
   INSTR(R2$,MID$(N$,2,1))=Ø OR INSTR
   (R1$,MID$(N$,3,1))=Ø THEN PRINT
   "ILLEGAL":GOTO 116Ø
1190 IF MID$(N$,4,1)="." THEN
   N$=LEFT$(N$,4) ELSE
   N$=LEFT$(N$,3)+"□"
1200 IF ST=NN THEN 123Ø
1210 FOR I=NN+1 TO ST+2 STEP -1
1220 N$(I)=N$(I-1):NEXT
1230 N$(ST+1)=N$
1240 ST=ST+1:NN=NN+1
1250 GOTO 115Ø
1260 CLS:PRINT"CHANGE WHICH
   NOTE";:INPUT ST
1270 IF ST=Ø THEN 94Ø
1280 IF ST>NN THEN ST=NN
1290 PRINT:PRINT"THIS IS CURRENTLY:":
   A$=N$(ST):RT=255:GOSUB 35Ø
1300 PRINTN$(ST)
1310 PRINT@32Ø:PRINT@32Ø,"";:INPUT
   "NEW CONTENTS:";N$
1320 IF LEN(N$)<3 THEN 131Ø
1330 IF INSTR(R3$,LEFT$(N$,1))=Ø OR
   INSTR(R2$,MID$(N$,2,1))=Ø OR INSTR
   (R1$,MID$(N$,3,1))=Ø THEN 131Ø
1340 IF MID$(N$,4,1)="." THEN N$=
   LEFT$(N$,4) ELSE N$=LEFT$(N$,3)+"□"
1350 N$(ST)=N$:FORK=1TO2ØØØ:NEXT:
   GOTO94Ø
1360 IF NN=Ø THEN RETURN ELSE
   CLS:PRINT@7,"LIST NOTES
   OPTION":C=Ø
1370 IF (PEEK(65314)AND1)=1 THEN 14ØØ
1380 POKE 329,255:PRINT@64,"";:
   INPUT "LIST TO PRINTER (Y/N)";P$
1390 IF P$="Y" THEN C=-2
1400 PRINT@128,"START AT NOTE
   NUMBER";:INPUT ST
1410 IF ST<=Ø THEN ST=1
1420 IF ST>NN THEN ST=NN
1430 CLS:LP=Ø
```

CUMULATIVE INDEX

An interim index will be published each week. There will be a complete index in the last issue of *INPUT*.

The publishers accept no responsibility for unsolicited material sent for publication in INPUT. All tapes and written material should be accompanied by a stamped, self-addressed envelope.

COMING IN ISSUE 45...

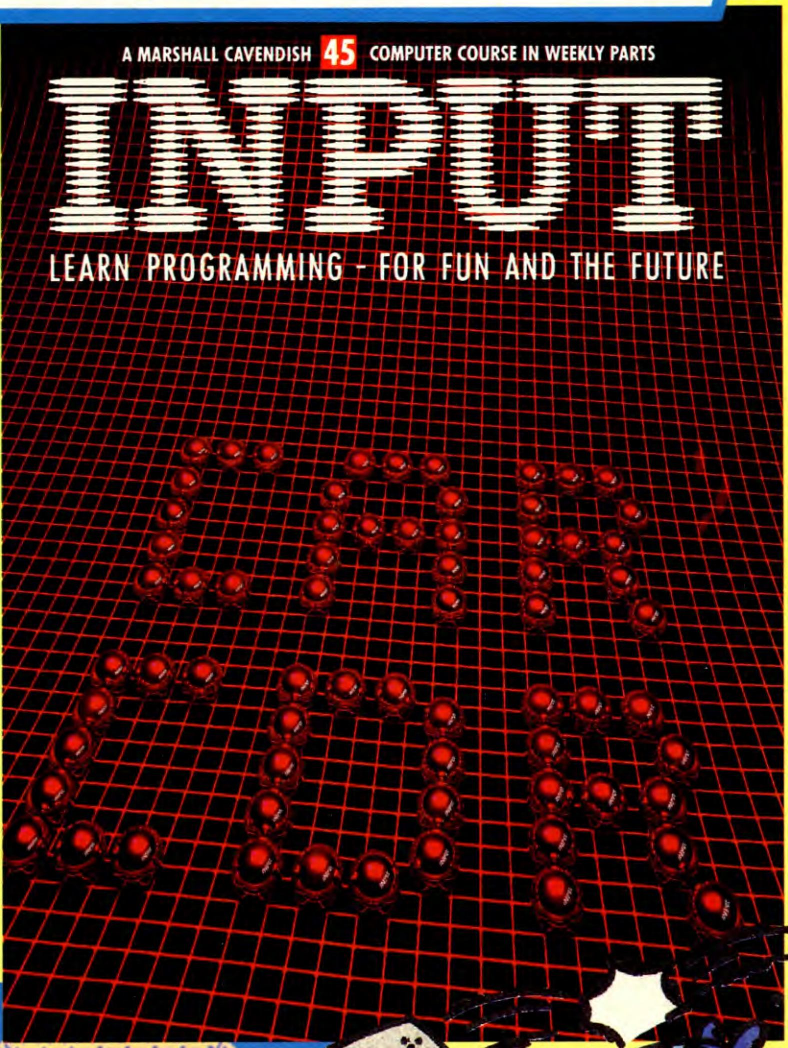
- ❑ *What have dragons, Albert Einstein, pretty pictures, pieces of string, and new concepts in mathematics and geometry in common? Read MODELS OF IRREGULARITY and find out!*

- ❑ *Budding Houdinis can start preparing to escape from the evil king's castle in part one of INPUT's new ADVENTURE GAME*

- ❑ *There at the beginning of computer languages, LISP is also the way of the future in many artificial intelligence applications*

- ❑ *Complete the MUSIC COMPOSER program and start dotting those minims, and tampering with tempo*

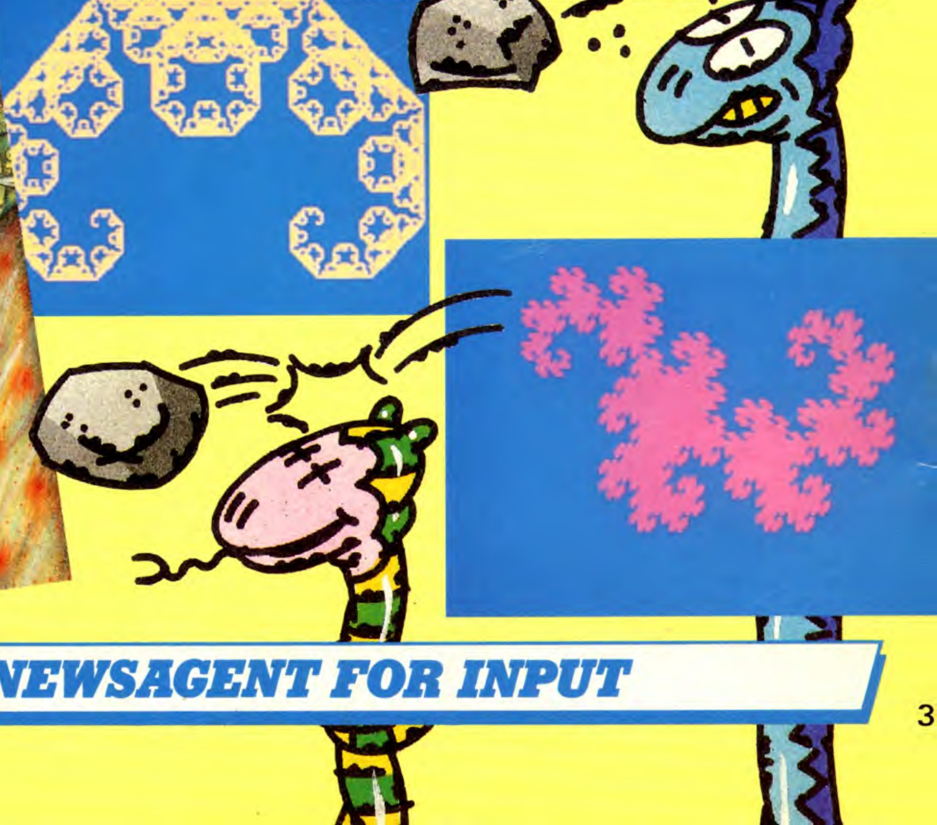
- ❑ *Willie whizzes out of the starting blocks and bounds up the hillside in the next part of CLIFFHANGER*



A MARSHALL CAVENDISH 45 COMPUTER COURSE IN WEEKLY PARTS

INPUT

LEARN PROGRAMMING - FOR FUN AND THE FUTURE



ASK YOUR NEWSAGENT FOR INPUT